RK0

The Real-Time Kernel '0'

System: v0.8.3-dev | Docbook: v083.1.0.1 | www.kernel0.org

Table of Contents

1. THE KERNEL AT A GLANCE . . . 1
   1.1. The design approach . . . 1
      1.1.1. Architecture . . . 1
      1.1.2. Programming with RK0 . . . 1
      1.1.3. Suitable Applications . . . 2
   1.2. Kernel Services . . . 2
2. **Task Scheduler** . . . 3
   2.1. Scheduler Data Structures . . . 3
      2.1.1. Task Control Block . . . 3
      2.1.2. Task Queues . . . 5
      2.1.3. Ready Queue Table . . . 5
      2.1.4. Waiting Queues . . . 5
      2.1.5. The scheduling algorithm . . . 6
   2.2. Handling the scheduler . . . 8
3. Timers and Delays . . . 10
   3.1. Busy-wait delay . . . 10
   3.2. Sleep Timers . . . 10
      3.2.1. Sleep Delay . . . 11
      3.2.2. Periodic Sleep . . . 12
   3.3. Blocking Time-out . . . 13
   3.4. Callout Timers (Application Timers) . . . 13
   3.5. System Tick . . . 14
   3.6. System Tasks . . . 14
4. Memory Allocator . . . 15
   4.1. How it works . . . 15
5. ***Inter-Task Communication*** . . . 17
   5.1. Task Flags . . . 17
      5.1.1. Usage Example: Supervisor Task . . . 17
   5.2. Semaphore . . . 19
      5.2.1. Counting Semaphore and Binary Semaphores . . . 19
      5.2.2. Semaphores in RK0 . . . 20
      5.2.3. Mutex Semaphore (Locks) . . . 22
         5.2.3.1. Priority Inversion and the Priority Inheritance Protocol . . . 23
      5.2.4. Mutexes vs Binary Semaphores . . . 27
   5.3. Scheduler Lock . . . 27
   5.4. Sleep Queue . . . 28
   5.5. Monitor Constructs . . . 29
      5.5.1. Monitor Invariant . . . 29

5.5.2. Signalling Discipline . . . 29
5.5.3. Producer-Consumer Solution using a Mesa Monitor . . . 30
5.5.4. Condition Variable Model . . . 32
5.5.4.1. Usage Example: Synchronisation Barrier . . . 33
5.5.4.2. Usage Example: Readers Writers Lock . . . 35
5.6. Message Queue . . . 38
5.6.1. Size of a Message . . . 39
5.6.2. Blocking and non-blocking behaviour . . . 39
5.6.3. Ownership and Priority Inversion . . . 39
5.6.4. Notify callback . . . 39
5.6.5. Usage Examples . . . 40
5.6.5.1. Averaging Sensor Values . . . 40
5.6.5.2. Queue Select using Notify Callback . . . 44
5.6.5.3. Synchronous Client-Server Procedure Call . . . 47
5.7. Most-Recent Message Protocol (MRM) . . . 49
5.7.1. Functional Description . . . 50
5.7.2. MRM Control Block Configuration . . . 51
5.7.3. Usage Example . . . 51
6. ***Error Handling*** . . . 55
6.1. Fail fast . . . 55
6.2. Stack Overflow . . . 55
6.3. Deadlocks . . . 56
7. ***RK0 Services API*** . . . 59

Chapter 1. THE KERNEL AT A GLANCE

1.1. The design approach

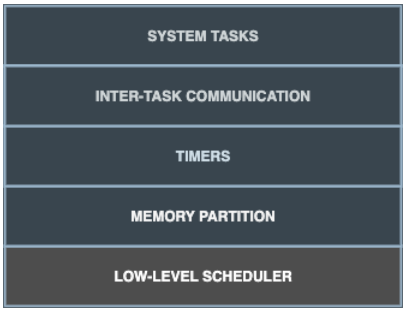
RK0 Blog: About Processes, Tasks and Threads

1.1.1. Architecture

The layered architecture can be split — roughly — into two: a top and a bottom layer. On the top, the *Executive* manages the resources needed by the application.

On the bottom, the *Low-level Scheduler* works as a software extension of the CPU.

Together, they implement the *Task* abstraction. This primitive is the *Concurrency Unit* and follows the *Thread* model. A *Task* is a *thread*.



In systems design jargon, the Executive enforces policy (what should happen). The Low-level Scheduler provides the mechanism (how it gets done). The services are the primitives that gradually translate policy decisions into concrete actions executed by the Scheduler. K0's goal is determinism on low-end devices.

Its multitasking engine operates without mimics of *userland*: tasks run in privileged mode on a different stack pointer from the system stack.

This trades the already limited 'isolation' its target architecture can provide, for tight and predictable control we need.

1.1.2. Programming with RK0

RK0 is designed so that it does not get in the programmer's way.

It aims to be transparent, composable, deterministic, and with clear semantics.

Transparent means that the kernel does not hide its primitives behind opaque pointers nor introduce unexpected side effects.

Its components are composable because each feature is self-contained yet designed to work with others.

Clear semantics means mechanims's behaviour are as expected accross different usage patterns.

Determinism is pursued by avoiding execution paths whose worst case cannot be bounded or reasoned about in advance.

When possible, operations are O(1), rely on word-aligned memory, and use static allocation.

When dynamic allocation is unavoidable, its worst-case behaviour can still be characterised offline (unless the application itself prevents any meaningful offline bound).

1.1.3. Suitable Applications

Given the architecture, *RK0* targets applications with the following characteristics:

1. They are designed to handle particular devices in which real-time responsiveness is imperative.
2. Applications and middleware may be implemented alongside appropriate drivers.
3. Drivers may even include the application itself.
4. *Untested programs are not loaded*: After the software has been tested, it can be assumed reliable.

1.2. Kernel Services

RK0 has *Core Services* (that cannot be disabled) and optional services (that can either be enabled or disabled).

Core Services:

- (Scheduler)
- Partition Memory Allocator
- Timer Delays
- Task Event Flags

Optional Services

- Application Timer (Callouts)
- Semaphore
- Mutex Lock
- Sleep Queue (Condition Variable)
- Message Queue (and its extensions)
- Most-Recent Message Buffer

When compiled solely with *Core Services* one gets a functional Executive with less than ~ 3KB ROM.

Chapter 2. Task Scheduler

RK0 employs a priority-based preemptive scheduler, aligned with a Rate-Monotonic Assignment. Tasks are typically assigned priorities according to their request rates - i.e., tasks with shorter periods are assigned to higher priorities. The highest priority is represented by the value '0'; the lowest is represented by the value '31'.

A scheduler remark is its constant time complexity (*O(1)*) and low latency. This was achieved by carefully composing the data structures along with an efficient *'choose-next'* algorithm. This is detailed below.

Time-slice was deprecated on version 0.5.0.

2.1. Scheduler Data Structures

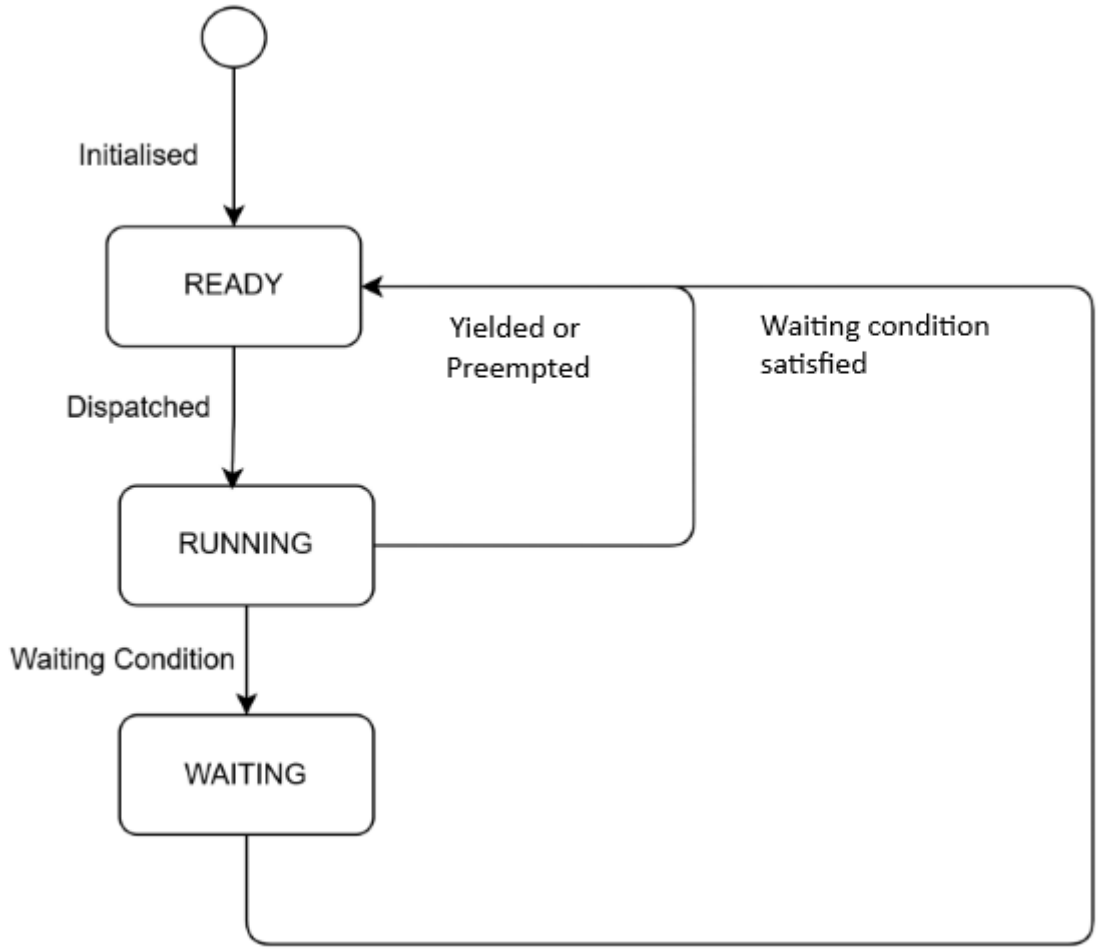
2.1.1. Task Control Block

Every primitive is associated to a data structure we refer to as its *Control Block*. A Task Control Block is a record for stack, resources, and time management. The table below partially represents a Task Control Block (as this document is live, this might not reflect the exact fields of the current version).

Task Control Block
Task name
Task ID
Status
Assigned Priority
Effective Priority
Saved Stack Pointer
Stack Address
Stack Size
Event Register Control Block
Last wake-time
Next wake-time
Time-out Flag
Preemption Flag
Owned Resources List
Waiting Resources List
Timeout List Node
TCB List Node

Tasks are static — they are not created (or destroyed) on runtime. There is no fork or join.

In practice, tasks are either *RUNNING* or '*waiting*' for their turn to run.



We need to define *WAITING* and *READY* clearly:

1. A *READY* task will be dispatched; therefore, switch to *RUNNING* whenever it is the highest priority *READY* task.
2. A *WAITING* task depends on a condition, generalised as an *event* to switch to *READY*.
3. Logically, the *WAITING* state will assume different pseudo-states related to the kind of event that will switch a task to *READY*:
    - *SLEEPING: a task is either sleeping (delayed) for a given time, or sleeping for a wake signal to be delivered to a _Sleep Queue*.
    - *PENDING*: the task suspended itself, waiting for a combination of Event Flags on its Event Register.
    - *BLOCKED*: A task is blocked on a mutex or semaphore.
    - *SENDING/RECEIVING*: A producer task, when blocking on a Message Passing object, switches its status to *SENDING*, and a consumer to *RECEIVING*.

> ***The scheduler rules, not the heap.***

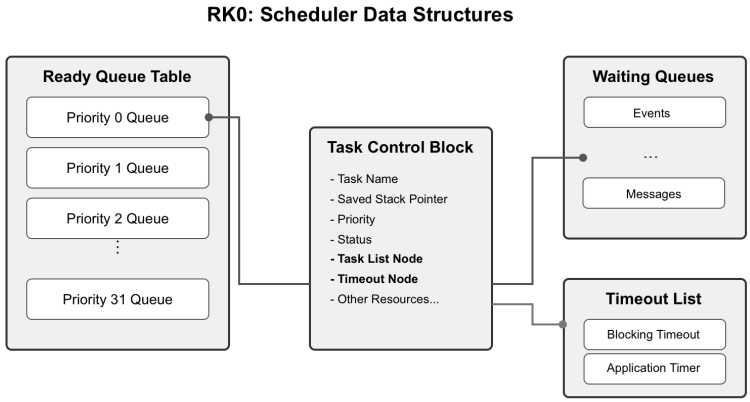
RK0 tasks are static.

It's a design decision rooted in real-time correctness.

Besides an application-specific system software does not need to treat tasks as 'unknown' objects.

The wins:

- A memory layout the systems programmer knows.
- No alignment traps.
- Link-time visibility:
  - Each task's stack is a named symbol in the linker map.
  - You can inspect and verify the memory layout before flashing.
  - A simple `objdump` reveals all stack allocations — that's peace of mind.



2.1.2. Task Queues

The backbone of the queues where tasks will wait for their turn to run is a circular doubly linked list: removing any item from a double list takes O(1) (provided we don't need to search the item). As the kernel knows each task's address, adding and removing is always O(1). Singly linked lists can't achieve O(1) for removal.

2.1.3. Ready Queue Table

Another design choice to achieve O(1) is the global ready queue, which is a table of FIFO queues—each queue dedicated to a priority—and not a single ordered queue. So, enqueuing a ready task is always O(1). Given the sorting needed, the time complexity would be O(n) if tasks were placed on a single ready queue.

2.1.4. Waiting Queues

The scheduler does not have a unique waiting queue. Every kernel object that can block a task has an associated waiting queue. Because these queues are a scheduler component, *they follow a priority discipline*: the highest priority task is dequeued first, *always*.

When an event capable of switching tasks from *WAITING* to *READY* happens, one or more tasks (depending on the mechanism) are then placed on the ready list, unique to their priority. Now, they are waiting to be picked by the scheduler—that is the definition of *READY*.

2.1.5. The scheduling algorithm

As the ready queue table is indexed by priority - the index 0 points to the queue of ready tasks with priority 0, and so forth, and there are 32 possible priorities - a 32-bit integer can represent the state of the ready queue table. It is a BITMAP:

```
The BITMAP computation: ((1a) OR (1b)) AND (2), s.t.:

(1a) Every Time a task is readied, update: BITMAP |= (1U << task->priority );
(1b) Every Time an empty READY QUEUE becomes non-empty, update: BITMAP |= (1U <<
queueIndex)
(2): Every Time READY QUEUE becomes empty, update: BITMAP &= ~(1U << queueIndex);
EXAMPLE:

  Ready Queue Index :     (6)5 4 3 2 1 0
          Not empty :      1 1 1 0 0 1 0
                           ------------->
                 (LOW)  Effective Priority  (HIGH)
In this case, the scenario is a system with 7 priority task levels. Queues with
priorities 6, 5, 4, and 1 are not empty.
```

Having the Ready Queue Table bitmap, we find the highest priority non-empty task list as follows:

(1) Isolate the **rightmost** '1':

```
RBITMAP = BITMAP & -BITMAP. (- is the bitwise operator for two's complement: ~BITMAP +
1) `
```

In this case:

```
                        [31]       [0]  :  Bit Position
                          0...1110010   :  BITMAP
                          1...0001110   : -BITMAP
                         =============
                          0...0000010   :  RBITMAP
                                 [1]
```

The rationale here is that, for a number N, its 2's complement -N, flips all bits - except the rightmost '1' (by adding '1') . Then, N & -N results in a word with all 0-bits except for the less significant '1'.

(2) Extract the **rightmost '1' *position***:

- For ARMv7M, we benefit from the `CLZ` instruction to count the *leading zeroes*. As they are the

number of zeroes on the left of the rightmost bit, '1', this value is subtracted from 31 to find the Ready Queue index.

```
RK_FORCE_INLINE static inline
unsigned __getReadyPrio(unsigned readyQBitmap)
{
    unsigned ret;
    __ASM volatile (
        "clz    %0, %1     \n"
        "neg    %0, %0     \n"
        "add    %0, %0, #31\n"
        : "=&r" (ret)
        : "r" (readyQBitmap)
        :
    );
    return (ret);
}
```

This instruction would return #30, and #31 - #30 = #01 in the example above.

- For ARMv6M there is no suitable hardware instruction. The algorithm is written in C and counts the *trailing zeroes*, thus, the index number. Although it might vary depending on your compiler settings, it takes ~11 cycles (*note it is still O(1)*):

```
/*
  De Brujin's multiply+LUT
  (Hacker's Delight book)
*/

/* table is on a ram section  for efficiency */
RK_SECTION(getReadyTable)
const static unsigned readyPrioTbl[32] =
{
 0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8,
 31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9
};

RK_FORCE_INLINE static inline
unsigned __getReadyPrio(unsigned readyQBitmap)
{
    unsigned mult = readyQBitmap * 0x077CB531U;

    /* Shift right the top 5 bits
     */
    unsigned idx = (mult >> 27);

    /* LUT */
    unsigned ret = (unsigned)readyPrioTbl[idx];
    return (ret);
```

```
}
```

For the example above, `mult = 0x2 * 0x077CB531 = 0x0EF96A62`. The 5 leftmost bits (the index) are `00001` → `table[1] = 1`.

During a context switch, the procedures to find the highest priority non-empty ready queue table index are as follows:

```
static inline RK_PRIO kCalcNextTaskPrio_(VOID)
{
    if (readyQBitMask == 0U)
    {
        return (idleTaskPrio);
    }
    readyQRightMask = readyQBitMask & -readyQBitMask;
    RK_PRIO prioVal = (RK_PRIO) (__getReadyPrio(readyQRightMask));
    return (prioVal);
}

VOID kSchSwtch(VOID)
{
    /* O(1) complexity */
    nextTaskPrio = kCalcNextTaskPrio_();

    RK_TCB* nextRunPtr = NULL;

    /* O(1) complexity */
    kTCBQDeq(&readyQueue[nextTaskPrio], &nextRunPtr);

    runPtr = nextRunPtr;

}
```

2.2. Handling the scheduler

> ***RK0 Blog:*** About Real-Time, Responsiveness and Throughput

An essential characteristic of the scheduler is that it is a *preemptive run-to-completion* scheduler. This term, '*run-to-completion*' has slightly different meanings depending on the context. It is often related to strictly cooperative schedulers, in the sense tasks must *yield* the processor. Otherwise, they monopolise the CPU.

In *RK0*, tasks with the same priority will work cooperatively. This is different from schedulers that employ a *time-slice* or a *quantum* for round-robin: after this time expires, task is put at the *tail* of the *Ready Queue*.

The term *run-to-completion* here is to be interpreted as follows:

- The scheduler's behaviour is to choose the highest priority READY task to run. Always.
- The scheduler works on a First-In-First-Out discipline for tasks with the same priority.
- A task must switch to the *READY* state before being eligible for scheduling.
- A task will switch from *RUNNING* to *READY* if yielding or if being preempted by a higher priority task. Otherwise it can only go to a *WAITING* state, and eventually switch back to *READY*.
- When a task is preempted by a higher priority task, it switches from *RUNNING* to *READY* and is placed back on the *head* position of its Ready Queue. This means that it will be resumed as soon as it is the highest priority ready task again.
- On the contrary, if a task *yields*, it tells the scheduler it has completed its cycle. Then, it will be enqueued on the ready queue tail - the last queue position.
- When a task *waits* it is suspended until a condition is satisfied.
- When the condition is satisfied, it switches from *WAITING* to *READY*, and is enqueued on the tail.
- So, tasks with the same priority cooperate by either *yielding* or *waiting*.
- If a task never yields or waits, other tasks with the same or lower priority *will starve*.
- Finally, Tasks with the same priority are *initially* placed on the *Ready Queue* associated with that priority in the order they are *created*.

RK0 can handle context-switching with an extended frame when a float-point co-processor is available. This must be informed when compiling by defining the symbol `__FPU_PRESENT=1`.

Chapter 3. Timers and Delays

3.1. Busy-wait delay

A busy-wait delay `kDelay(t)` keeps a task spinning for `t` ticks. That is, the task does nothing but does not suspend or yield (but can be preempted). This service finds its use when simulating workloads.

Context switching is probably the most significant overhead on a kernel. The time spent on the System Tick handler contributes to much of this overhead.

Design Choice:

- Timers are kept on a single list; only the head element needs to be updated using a delta-queue approach.
- Application Timers that trigger callbacks are run on a deferred, non-preemptible system task.

Benefits:

- Keep the overhead of updating timers as minimal as possible with the delta queue;
- Deferring the Application Timer to a high-priority, non-preemptible system task meet the requested callback period while keeping the ability to track system ticks.

Timeout Node
Timeout Type
Absolute Interval (Ticks)
Relative Interval (Ticks)
Waiting Queue Address
Next Timeout Node
Previous Timeout Node

Every task is prone to events triggered by timers described in this section. Every Task Control Block has a node to *a timeout list*. This list is doubly linked treated a delta-sequence.

A set T_{set} `= {(T1,8), (T2,6), (T3,10)}` will be started at a relative time `0` as a sequence T_{seq} `= <(T2,6), (T1,2), (T3,2)>`.

Thus, for every system tick, only the head element on the list needs to be decreased — yielding O(1).

3.2. Sleep Timers

A task can be suspended by an amount of time in ticks, in two distinct manners:

3.2.1. Sleep Delay

The task sleeps for the exact number of `t` ticks on every call. Time elapsed between calls is not considered.

Example:

```
VOID Task1(VOID* args)
{
    RK_UNUSEARGS
    UINT count = 0;
    while (1)
    {

        logPost("Task1: sleep");
        kSleep(300);
        /* wake here */
        count += 1U;
        if (count >= 5)
        {
            kDelay(25); /* spin */
            count=0;
            /* every 5 activations there will be a drift */
        }
    }
}
```

Output:

```
0 ms :: Task1: sleep
300 ms :: Task1: sleep  <-- +300
600 ms :: Task1: sleep  <-- +300
900 ms :: Task1: sleep  <-- +300
1200 ms :: Task1: sleep <-- +300
1525 ms :: Task1: sleep <-- +325
1825 ms :: Task1: sleep <-- +300
2125 ms :: Task1: sleep <-- +300
2425 ms :: Task1: sleep
2725 ms :: Task1: sleep
3050 ms :: Task1: sleep
3350 ms :: Task1: sleep
3650 ms :: Task1: sleep
3950 ms :: Task1: sleep
4250 ms :: Task1: sleep
4575 ms :: Task1: sleep
```

3.2.2. Periodic Sleep

This primitive is intended to create *periodic activations*. The period `P` ticks is defined at the first kernel call `sleepperiod(P)`, and adjusted internally on subsequent activations, as follows:

Say a task is expected to return from its k_{eth} sleep at $T_{k+1} = T_k + P$ [ticks]. If the task is resumed at $T_{k+1} = T_k + P + N$, upon detecting this drift, the kernel sets: $(T_{k+2} = T_{k+1} + P - N)$.

This can be rewritten as:

$(T_{k+2} = T_k + P + N + P - N) \leftrightarrow (T_{k+2} - T_k = 2P)$

Example:

```
VOID Task1(VOID* args)
{
    RK_UNUSEARGS
    UINT count = 0;
    while (1)
    {

        logPost("Task1: sleep periodic");
        kSleepPeriod(300); /*P=300 ticks; tick=1ms*/
        /* wake here */
        count += 1U;
        if (count >= 5)
        {
            kDelay(25); /* spin */
            count=0;
            /* every 5 activations there will be a drift */
        }
    }
}
```

Output:

```
.
.

1200 ms :: Task1: sleep periodic (4P)    (n)
1525 ms :: Task1: sleep periodic (>5P)    |
1800 ms :: Task1: sleep periodic (6P)     |
2100 ms :: Task1: sleep periodic (7P)     |
2400 ms :: Task1: sleep periodic (8P)     |
2700 ms :: Task1: sleep periodic (9P)     |
3025 ms :: Task1: sleep periodic (>10P)   |
3300 ms :: Task1: sleep periodic (11P)    |
3600 ms :: Task1: sleep periodic (12P)    |
3900 ms :: Task1: sleep periodic (13P)    |
4200 ms :: Task1: sleep periodic (14P)    |
```

```
4525 ms :: Task1: sleep periodic (>15P)   |
4800 ms :: Task1: sleep periodic (16P)   (m)  m-n=12
.                                       -----
.                                  Phase=3600=12xP
.
```

3.3. Blocking Time-out

These are internal timers associated with kernel calls that are blocking. Thus, establishing an upper-bound waiting time might benefit them. When the time for unblocking is up, the kernel call returns, indicating a timeout. This value is passed as a number of ticks.

When blocking is associated with a kernel object (other than the Task Control Block), the timeout node will store the object waiting for queue's address, so it can be removed if time expires.

A kernel call is made non-blocking, that is *try semantics*, by assigning the value `RK_NO_WAIT`, the function returns immediately if unsuccessful. The value `RK_WAIT_FOREVER` suspends a task indefinitely until the condition is satisfied.

In practice, we often block either using `RK_WAIT_FOREVER` or do not block (*try semantics*, `RK_NO_WAIT`).

Use a bounded timeout only when you expect occasional misses and you know how to handle them. If a blocking call times out and no recovery plan is feasible, it is as a system fault (on constrained devices this is usually unrecoverable at runtime; a watchdog is what is left).

Importantly, *an ISR shall **never** blocks*. Indeed, any blocking call from an ISR will hard fault if error checking is enabled.

3.4. Callout Timers (Application Timers)

Timer Control Block
Option: Reload/One-Shot
Phase (Initial Delay)
Callout Function Pointer
Callout Argument
Timeout Node

These are Application Timers that will issue a callback when expiring. In addition to a callout function, an Application Timer receives an initial phase delay and a period and can choose to run once (one-shot) or auto-reload itself.

The callback runs within a System Task with priority 0 and is non-preemptible, which makes the scheduler prioritise it over other tasks. Callouts must be short and unblocking, as they can cause high CPU contention.

For clarity, Timer Callouts are on a separate list in the kernel, although they share the same `TIMEOUT`

node.

Application Timers (with autoreload) will keep track of delays in between activations, to preserve phase accross calls as in Periodic Sleep

Callout Timers usage are found throughout this docbook to emulate interrupts.

3.5. System Tick

A dedicated peripheral that generates an interrupt after a defined period provides the kernel time reference. For ARMv6/7M, this peripheral is the built-in SysTick, a 24-bit counter timer. The handler performs some housekeeping on every tick and assesses the need to call a context switch.

The 'housekeeping' accounts for global timer tracking and any tick-dependent condition that might change a task status. When a timer expires, it might switch a task from `WAITING` to `READY` or dispatch a callback. In the case of a callback, this will also trigger a context-switching for the Post-Processing Handler (a System Tasks) in which the callback is executed and the related timer(s) are appropriately updated.

Note that tasks might switch from `WAITING` to `READY` for reasons other than tick-related. In these cases, context switching might be triggered immediately if the readied task can preempt the running task.

3.6. System Tasks

There are two *System Tasks*: the *Idle Task* and the *Post-Processing Task*.

The *Post-Processing Task* is is today used solely for Timer callouts, then normarly refered with a label `TIMHANDLE` for configurations. Nominally its priority is `0`, but on practice it could be considered as having priority `-1`, because it always takes precedence over other tasks with priority `0`. *It can't be preempted* by any user task, so again, timer callouts need to be short. If you are using a application timer with reload for heavier processing than a 'time-mark', what you need is a periodic task using the `kSleepPeriodc()` or the callout will itself defer the processing to another task.

The *Idle Task* runs whenever there is no other ready task to be dispatched. The CPU enters on *low-power*. The kernel assigns the *Idle Task* priority during initialisation, taking into account all priorities the user has defined. Unless user tasks occupy all 32 priorities, the Idle Task is treated as an ordinary lowest priority and has a position in the ready queue table. Otherwise, it is selected if *Ready Queue Bitmap* is `0x00000000`.

Chapter 4. Memory Allocator

Memory Allocator Control Block
Associated Block Pool
Number of Blocks
Block Size
Number of Free Blocks
Free Block List

The standard C library `malloc()` leads to fragmentation and (also, because of that) is highly indeterministic. Unless we use it once - to allocate memory before starting up, it doesn't fit. But often, we need to 'multiplex' memory amongst tasks over time, that is, to dynamically allocate and deallocate.

To avoid fragmentation, we use fixed-size memory blocks. A simple approach would be a static table marking each block as free or taken. With this pattern, you will need to 'search' for the next available block, if any - the time for searching changes - bounding this search to a maximum number of blocks, or *O(n)*. To optimise, an approach is to keep track of what is free using a dynamic table—a linked list of addresses. Now we have *O(1)*.

We use "meta-data" to initialise the linked list. Every address holds the "next" address value. All addresses are within the range of a pool of fixed-size blocks. This approach limits the minimal size of a block to the size of a memory address—32 bits for our supported architecture.

Yet, this is the cheapest way to store meta-data. If not stored on the empty address itself, an extra 32-bit variable would be needed for each block, so it could have a size of less than 32 bits.

Allocating memory at runtime is a major source of latency (1), indeterministic (2) behaviour, and footprint overhead (3).

Design choice: the allocator's design achieves low-cost, deterministic, fragmentation-free memory management by using fixed-size word-aligned block sizes (1)(2) and embedding metadata within the memory blocks themselves (3).

Benefits: Run-time memory allocation benefits have no real-time drawbacks.

Importantly, the kernel will always round up the block size to the next multiple of 4. Say the user creates a memory pool, assigning blocks to be 6-byte wide; they will turn into 8-byte blocks.

4.1. How it works

When a routine calls `alloc()`, the address to be returned is the one a "free list" is pointing to, say `addr1`. Before returning `addr1` to the caller, we update the free list to point to the value stored within `addr1` - say `addr8` at that moment.

When a routine calls `free(addr1)`, we overwrite whatever has been written in addr1 with the value-

free list point to (if no more `alloc()` were issued, it would still be `addr8`), and `addr1` becomes the free list head again.

Allocating and deallocating fixed-size blocks using this structure and storing meta-data this way is as deterministic (*O(1)*) and economical as we can get for dynamic memory allocation.

A drawback is if a routine writes to non-allocated memory within a pool it will spoil the meta-data and the Allocator will fail.

Chapter 5. *Inter-Task Communication*

RK0 Blog:

- About Inter-Task Communication - Part 1
- About Inter-Task Communication - Part 2

Inter-Task Communication (ITC) refers to the mechanisms that enable tasks to coordinate/cooperate/synchronise by means of sending or receiving information that falls into two logical categories: *Signals* or *Messages*.

- ***Signals***: A *Signal is either present or absent*. It is meaning is implicit.
- ***Messages***: A *Message* is a means of coordinating and exchanging information altogether. Different from a *Signal*, a Message needs to be parsed (interpreted).

5.1. Task Flags

Within Task Control Block
Event Register (32 flags)
Required Signal Flags
Options (ALL/ANY)

Each Task Control Block stores a 32-bit Event Register, representing a combination of 32 different events, if defining 1 event/bit. A bit set means an event notification is pending to be detected.

Bitwise friendly, the API is written as set() (as to signal/post), get() (as to wait/pend).

A task checks for a combination of events it is expecting. This combination can be satisfied if ANY (OR logic) of the required bits are set or if ALL of the required bits are set (AND logic).

Thus, if the condition is not met the task can optionally suspends, switching to the logical state PENDING.

When another task issues a set() which result satisfies the waiting condition, the task state is then READY.

Upon returning, all required positions have been cleared on the Task's Event Register.

A set is always an OR operation of an input mask over the current value. 0x00 is invalid for both set() and get() operations.

Additional operations are to query a tasks's event register, and to clear its own registers.

5.1.1. Usage Example: Supervisor Task

One possible usage pattern is a task's cycle begins checking for any events (it is able/supposed to

handle). If using it on a supervisor task — it can create a neat event-driven pattern for a soft/firm real-time system:

```
typedef struct
{
    ULONG pendingBit;
    TaskHandle_t dstTask;
    ULONG dstSignal;
} Route_t;

static const Route_t routes[] =
{
    {
        PENDING_AIRFLOW_INCREASE,
        airFlowTaskHandle,
        AIRFLOW_INCREASE_SIGNAL
    },
    {
        PENDING_TEMP_DECREASE,
        tempTaskHandle,
        TEMP_DECREASE_SIGNAL
    },
    /* more routes */
}

VOID SupervisorTask(VOID *args)
{
    RK_UNUSEARGS;

    while(1)

    {
        ULONG gotFlags = 0UL;

        RK_ERR err = kTaskFlagsGet(0xFFFF,
                                   RK_FLAGS_ANY,
                                   &gotFlags,
                                   SUPERVISOR_T_PERIOD);

        if (err == RK_ERR_SUCCESS && gotFlags != 0)
        {
            for (ULONG i = 0; i < ARRAY_LEN(routes); ++i)
            {
                if (gotFlags & routes[i].pendingBit)
                {
                    kTaskFlagsSet(routes[i].dstTask, routes[i].dstSignal);
                }
            }
        }
```

```
        /* if there is anything to do if time out */
    }
}
```

Task Signals are the the only ITC primitive that cannot be disabled, thus, they are regarded as a *Core Mechanism*.

RK0 does not implement BSD/UNIX-like asynchronous signals (POSIX style).

Tasks explicitly wait (`get()`) for bit patterns, and others `set()` bit patterns, making the scheduler aware of all blocking conditions.

Thus, unlike traditional asynchronous signal mechanisms (as BSD/UNIX Signals), no code is ever executed at an arbitrary instruction boundary.

5.2. Semaphore

Semaphore Control Block
Counter (Unsigned Integer)
Maximum Value
Waiting Queue

A semaphore S is a nonnegative integer variable, apart from the operations it is subjected to. S is initialized to a nonnegative value. The two operations, called P and V, are defined as follows:

`P(S): if S > 0 then S := S-1, else the process is suspended until S > 0.`

`V(S): if there are processes waiting, then one of them is resumed; else S := S+1.`

(Dijkstra, 1968)

Semaphores are *public* kernel objects for signalling and waiting for events.

`V()` in RK0 semaphores maps to `post` and `P()` to `pend()`.

5.2.1. Counting Semaphore and Binary Semaphores

The typical use case for Semaphores is as a "credit tracker" — one uses it to verify (wait/pend) and indicate (signal/post) the availability of a countable resource — say, the number of slots within a queue. These are Counting Semaphores.

The so-called Binary Semaphore is just a counting semaphore that counts up to 1 — they do not accumulate; thus, it is either AVAILABLE or UNAVAILABLE (also FULL or EMPTY).

Binary Semaphores are employed for task-to-task (unilateral or bilateral) or ISR-to-task (unilateral) synchronisation, as well as for mutual exclusion, in which case there are some drawbacks.

5.2.2. Semaphores in RK0

To initialise a Semaphore in RK0 , two values are needed: the initial and the *maximum*. When the *maximum* value is reached, the counter is not incremented anymore and the operation returns (RK_ERR_SEMA_FULL).

Thus, a *Binary Semaphore* is created by setting its maximum value to 1. A counting semaphore that will possibly never get full is set to its maximum value as `UINT32_MAX`.

Besides initialisation, `post()` and `pend()`, a `query()` operation inspects the semaphore status (a non-negative value is the semaphore counter, a negative value is the number of tasks waiting on the semaphore).

A `flush()` operation, is a *broadcast* signal. It releases all tasks pending on that semaphore.

Bounded Buffer with Semaphores

Data (items) are a buffered within a memory region that capacity is `K` items.

Thus: `0 ≤ (Number of Inserted) − (Number of Extracted) ≤ K`.

Using semaphores the pattern is as follows:

1. A semaphore with `K` tokens to track the number of free slots, not allowing producers to proceed if there no free slots.
2. Another semaphore, with `K` tokens, for the number of items, not allowing consumers to proceed if there are no items.
3. A 1-token semaphore, so a single task can manipulate the buffer at at a time.

```
/* a ring buffer of items */
#define BUFSIZ (K)
static ITEM_t buf[BUFSIZ]={0};
static UINT getIdx = 0U;
static UINT putIdx = 0U;
/* this indexes==0 could either mean FULL or EMPTY for a regular
circular buffer with wrap-around.
With semaphores the state is well defined.
*/

RK_SEMAPHORE  itemSema;
RK_SEMAPHORE  slotSema;
RK_SEMAPHORE  acquireSema;


VOID kApplicationInit(VOID)
{

  /*buffer is initialised empty */
    kSemaphoreInit
    (   &itemSema,
```

```
        0,   /* no item  */
        K    /*max items */
    );

    kSemaphoreInit
    (   &slotSema,
        K, /* K free slots */
        K  /* max slots */
    );

    /* and free */
    kSemaphoreInit
    (   &acquireSema,
        1, /* free to access */
        1  /* 1 max task allowed */
    );


VOID PutItem(ITEM_t* insertItemPtr)
{
    /* wait for room */
    kSemaphorePend(&slotSema, RK_WAIT_FOREVER);

    /* wait for availability */

    kSemaphorePend(&acquireSema,  RK_WAIT_FOREVER);
    buf[putIdx] = *insertItemPtr;
    putIdx += 1U; putIdx %= BUFSIZ;
    /* signal availability */
    kSemaphorePost(&acquireSema);

    /* signal item */
    kSemaphorePost(&itemSema);
}


 VOID GetItem(ITEM_t* extractItemPtr )
{

    /* wait for an item */
    kSemaphorePend(&item, RK_WAIT_FOREVER);

    /* wait for availability */
    kSemaphorePend(&acquireSema,  RK_WAIT_FOREVER);

    *extractItemPtr = buf[getIdx];
    getIdx+=1U; getIdx %= BUFSIZ;

    /* signal availability */
    kSemaphorePost(&acquireSema);
```

```
    /* signal room */
    kSemaphorePost(&slotSema);
}
```

The solution above has Put() and Get() as blocking methods.

If the producer and the consumer run at different rates, eventually, they will synchronise to the lowest rate.

The numbers below are from a run with a buffer of 32 items (integers being incremented are the produced data).

The producer is twice faster than the consumer. Initially at every 2 insertions there is a single remove. Eventually, the buffer is filled up, and tasks run at lockstep (put, get, put, get...), at the consumer pace:

```
Put 59 <-
Put 60 <-
------
Got 30 ->
------
Put 61 <-
Put 62 <-
------
Got 31 ->
------
Put 63 <-
Put 64 <-
--------
Got 32  | ->
Put 65  . <-
       <x>[Full Queue, Producer blocks]
Got 33  | ->  [Consumer unblocks producer...]
Put 66  . <-
       <x>[Full Queue]
Got 34  | ->  [Consumer unblocks producer...]
Put 67  . <-
       <x>[Full Queue]
```

5.2.3. Mutex Semaphore (Locks)

Mutex Control Block
Locked State (Boolean)
Owner
Waiting Queue

Mutex Control Block
Mutex Node (list node within the owner TCB)

Some code regions are critical in that they cannot be accessed by more than one task at once. Acquiring (`lock()`) a mutex before entering a region and releasing it when leaving makes that region mutually exclusive.

A Mutex is another semaphore specialisation — it can be seen as a binary semaphore with a notion of ownership - when a task succesfully acquires a mutex is now the *owner*, and only this task can release it.

If a task tries to acquire an already locked mutex, it switches to `BLOCKED` state until *the mutex is unlocked by its owner*. Then, the highest priority task waiting to acquire the resource is dequeued, as on semaphores.

However, unlike semaphores, the complementary operation, `unlock()`, when issued by a non-owner, has undefined behaviour. In K0, it will be a hard fault.

Mutexes are solely for mutual exclusion; they cannot be used for signalling. It is common to use Counting Semaphores initialised as 1, or Binary Semaphores for mutual exclusion.

However, particularly for a Counting Semaphore, if the count increases twice in a row, the mutual exclusion is gone. For both, *Priority Inversion* can become a problem, as will be explained.

PS: Mutexes in RK0 are not recursive. One cannot make reentrant calls on critical regions.

5.2.3.1. Priority Inversion and the Priority Inheritance Protocol

Let TH, TM, and TL be three tasks with priority high (H), medium (M) and low (L), respectively. Say TH is dispatched and blocks on a mutex that 'TL' has acquired (i.e.: *"TL is blocking TH"*).

If 'TM' does not need the resource, it will run and preempt 'TL'. And, by transition, 'TH'.

From now on, 'TH' has an *unbounded waiting time* because any task with priority higher than 'L' that does not need the resource indirectly prevents it from being unblocked — *awful.*

The Priority Inheritance (PI) Protocol avoids this unbounded waiting. It is characterised by an invariant, simply put:

> *At any instant a Task assumes the highest priority amongst the tasks it is blocking.*

If employed on the situation described above, task TM cannot preempt TL, whose effective priority would have been raised to 'H'.

It is straightforward to reason about this when you consider the scenario of a single mutex.

But when locks nest — that is, more than one critical region — the protocol also needs to be:

- Transitive: that is, if T1 is blocking T2, and T2 is blocking T3, if T3 has the highest priority, T3 propagates its priority to T1 via T2.
- A task can own several mutexes at once. Thus, when exiting the critical region it needs to look up each waiting queue, and assume the highest priority. If there are no blocked tasks behind, its nominal priority is then restored. (As tasks are enqueued by priority, it means looking at the task waiting on the head of each waiting queue.)

Below, a demonstration:

```
/* Task1 has the Highest nominal priority */
/* Task2 has the Medium nominal priority */
/* Task3 has Lowest nominal priority */

/* Note Task3 starts as 1 and 2 are delayed */

RK_DECLARE_TASK(task1Handle, Task1, stack1, STACKSIZE)
RK_DECLARE_TASK(task2Handle, Task2, stack2, STACKSIZE)
RK_DECLARE_TASK(task3Handle, Task3, stack3, STACKSIZE)


RK_MUTEX mutexA;
RK_MUTEX mutexB;

VOID kApplicationInit(VOID)
{
    K_ASSERT(!kCreateTask(&task1Handle, Task1, RK_NO_ARGS, "Task1", stack1, \
        STACKSIZE, 1, RK_PREEMPT));
    K_ASSERT(!kCreateTask(&task2Handle, Task2, RK_NO_ARGS, "Task2", stack2, \
        STACKSIZE, 2, RK_PREEMPT));
    K_ASSERT(!kCreateTask(&task3Handle, Task3, RK_NO_ARGS, "Task3", stack3, \
        STACKSIZE, 3, RK_PREEMPT));

/* mutexes initialised with priority inheritance enabled */
    kMutexInit(&mutexA, RK_INHERIT);
    kMutexInit(&mutexB, RK_INHERIT);
}



VOID Task3(VOID *args)
{
    RK_UNUSEARGS
    while (1)
    {
        printf("@ %lums: [TL] Attempting to LOCK 'A' | Eff: %d | Nom: %d\r\n",
kTickGet(),
               runPtr->priority, runPtr->prioReal);

        kMutexLock(&mutexA, RK_WAIT_FOREVER);
```

```
        printf("@ %lums: [TL] LOCKED 'A' (in CS) | Eff: %d | Nom: %d\r\n", kTickGet(),
               runPtr->priority, runPtr->prioReal);

        kDelay(60); /* <-- important */

        printf("@%lums: [TL] About to UNLOCK 'A' | Eff: %d | Nom: %d\r\n", kTickGet(),
               runPtr->priority, runPtr->prioReal);

        kMutexUnlock(&mutexA);

        printf("--->");
        printf("@%lums: [TL] Exit CS | Eff: %d | Nom: %d\r\n", kTickGet(),
               runPtr->priority, runPtr->prioReal);

        kSleep(4);
    }
}

VOID Task2(VOID *args)
{
    RK_UNUSEARGS
    while (1)
    {
        kSleep(5);

        printf("@%lums: [TM] Attempting to LOCK 'B' | Eff: %d | Nom: %d\r\n",
kTickGet(),
               runPtr->priority, runPtr->prioReal);
        kMutexLock(&mutexB, RK_WAIT_FOREVER);

        printf("@%lums: [TM] LOCKED 'B', now trying to LOCK 'A' | Eff: %d | Nom:
%d\r\n",
               kTickGet(), runPtr->priority, runPtr->prioReal);
        kMutexLock(&mutexA, RK_WAIT_FOREVER);

        printf("@%lums: [TM] LOCKED 'A' (in CS) | Eff: %d | Nom: %d\r\n", kTickGet(),
               runPtr->priority, runPtr->prioReal);
        kMutexUnlock(&mutexA);

        printf("@%lums: [TM] UNLOCKING 'B' | Eff: %d | Nom: %d\r\n", kTickGet(),
               runPtr->priority, runPtr->prioReal);

        kMutexUnlock(&mutexB);

        printf("--->");

        printf("@%lums: [TM] Exit CS | Eff: %d | Nom: %d\r\n", kTickGet(),
               runPtr->priority, runPtr->prioReal);
    }
}
```

```
VOID Task1(VOID *args)
{
    RK_UNUSEARGS
    while (1)
    {
        kSleep(2);

        printf("@%lums: [TH] Attempting to LOCK 'B'| Eff: %d | Nom: %d\r\n",
kTickGet(),
               runPtr->priority, runPtr->prioReal);

        kMutexLock(&mutexB, RK_WAIT_FOREVER);

        printf("@%lums: [TH] LOCKED 'B' (in CS)  | Eff: %d | Nom: %d\r\n", kTickGet(),
               runPtr->priority, runPtr->prioReal);

        kMutexUnlock(&mutexB);

        printf("--->");

        printf("@%lums: [TH] Exit CS | Eff: %d | Nom: %d\r\n", kTickGet(),
               runPtr->priority, runPtr->prioReal);
    }
}
```

Result and comments:

```
>>>> TL locks 'A'. Higher priority tasks are sleeping. <<<<

@ 14720ms: [TL] Attempting to LOCK 'A' | Eff: 3 | Nom: 3
@ 14720ms: [TL] LOCKED 'A' (in CS) | Eff: 3 | Nom: 3

@14721ms: [TM] Attempting to LOCK 'B' | Eff: 2 | Nom: 2

>>>> TM acquires 'B' and is blocked by TL on 'A'. TL inherits TM's  priority. <<<<

@14721ms: [TM] LOCKED 'B', now trying to LOCK 'A' | Eff: 2 | Nom: 2

>>>> TH will blocked by TM on 'B': <<<<

@14722ms: [TH] Attempting to LOCK 'B'| Eff: 1 | Nom: 1

>>>> TM inherits TH's priority. TL inherits TH's priority via TM. <<<<

@14780ms: [TL] About to UNLOCK 'A' | Eff: 1 | Nom: 3

>>>> Upon unlocking 'A', TL is preempted by TM. It means TL's priority has been
restored, as it is no longer blocking a higher priority task. <<<<
```

```
>>>> Now TM acquires 'A' <<<<

@14780ms: [TM] LOCKED 'A' (in CS) | Eff: 1 | Nom: 2

>>>> After releasing 'A', but before releasing 'B', TM's priority is still '1', as it
is blocking TH while holding 'B'. <<<<

@14780ms: [TM] UNLOCKING 'B' | Eff: 1 | Nom: 2

>>>> Upon unlocking 'B' TM is preempted by TH. (TM's priority has been restored.) <<<<

@14780ms: [TH] LOCKED 'B' (in CS)  | Eff: 1 | Nom: 1

>>> RESULT: even though priority inversion was enforced, tasks leave the nested lock
ordered by their nominal priority. <<<

--->@14780ms: [TH] Exit CS | Eff: 1 | Nom: 1
--->@14780ms: [TM] Exit CS | Eff: 2 | Nom: 2
--->@14780ms: [TL] Exit CS | Eff: 3 | Nom: 3
```

Importantly, the worst-case time is bounded by the time the lowest priority task holds a lock (60 ms in the example: 14720ms → 14780ms).

As for each priority update we check each waiting queue for each mutex a task owns, the time-complexity is is linear O(owner*mutex). But, typically no task ever holds more than a few mutexes. Yet, one should not be encouraged to nest locks if not needed.

5.2.4. Mutexes vs Binary Semaphores

There is (or used to be) a lot of fuss about whether binary semaphores are appropriate to use as locks. As a practical guideline, if all tasks sharing the resource have the same priority, using a binary semaphore *can be appropriate* — because a binary semaphore is considerably faster. It all depends on the case.

The drawback is the lack of ownership: any task can accidentally release the resource. On a large codebase, this can become a real problem. Nonetheless, this is a problem for semaphores in general.

For tasks with different priorities, binary semaphores should never be considered for mutual exclusion unless priority inversion is not a problem (how?).

Counting semaphores initialised as 1 is too risky. Besides the priority inversion, if the count ever increases above 1, mutual exclusion is lost, and multiple tasks can enter the critical section at once.

5.3. Scheduler Lock

Often, we need a task to perform operations without being preempted. A mutex serialises access to a code region but does not prevent a task from being preempted while operating on data. Depending on the case, this can lead to inconsistent data state.

An aggressive way is to disable interrupts globally. For kernel services often it is the only way to keep data integrity. On the higher level it is feasible for very short operations and/or when you need to protect data from interrupts altogether.

A less aggressive approach is to make the task non-preemptible with `kSchLock()` before entering the critical region and `kSchUnlock()` when leaving. This way, interrupts are still being sensed, and even higher-priority tasks might switch to a ready state, but the running thread will not be preempted.

The priority inversion it potentially causes is bounded. If a higher-priority task is readied while the scheduler is locked, the context switch happens immediately after unlocking.

Note that for locking/unlocking the scheduler the global interrupts will be disabled for the time to increment/decrement a counter, therefore, if your atomic operation is as short as that (3 to 4 cycles), disabling/enabling global interrupts is a better alternative.

To add to the discussion, when two threads need to access the same data to 'read-modify-write', a lock-free mechanism is the LDREX/STREX operations of ARMv7M (or more generally C11 atomics). They do not avoid preemptions, and particularly in ARMv7m, if the data is touched by an ISR before the store-exclusive concludes, the ownership is lost. Typically used for multi-core spin-locking.

5.4. Sleep Queue

Sleep Queue Control Block
Task Waiting Queue

The `RK_SLEEP_QUEUE` object is simply a queue of tasks sleeping waiting for a signal/wake operation on them. That could be read 'as tasks sleeping, until they are signalled an event has happened'.

That's why this primitive was formerly called `RK_EVENT`. Naturally we might name the queue as to indicate the event, which normally is a state (e.g., `notFull`, `notEmpty`) or the action it triggers (e.g., `goWriters`, `goReaders`).

An *RK_SLEEP_QUEUE* object does not have any records to indicate if an associated event has ever happened. Thus, a call `wait(&sleepq, timeout)` *always* put the caller task to sleep. Note that using `RK_NO_WAIT` on this primitive is meaningless, because there is nothing to 'try'. The call will just return.

A `signal(&sleepq)` will wake-up a single task - the highest priority. A `wake(&sleepq, n, &r)` is a *broadcast*: at most `n` sleeping tasks will switch to `READY`. `r` will store the number of remaining tasks, if any.

If willing to wake *all* tasks, one either make `n=0`, or use the `flush(&sleepq)` helper.

A `query(&sleepq)` operation returns the number of sleeping tasks.

Unique to Sleep Queues is the operation `suspend(&sleepq, taskHandle)` that can be used either by a

`RUNNING` task (or ISR) to remove a task from the ready queue when we do not want it to be scheduled, or less commonly by an ISR to suspend the `RUNNING` task it has interrupted. Anyway, if you find yourself manually removing tasks from the `READY` queue often, it is a code smell.

The operation `ready(&sleepq, taskHandle)` is the opposite of `suspend` and removes an specific task from a sleep queue, moving it to the ready queue.

It is important to note that the *stateless* characteristic of Sleep Queues make it a very limited mechanism to be used alone, as they are prone to *lost wake-up signals*. *There are suitable cases*, all *simple unilateral cases*. Yet, its main purpose is as a building block for *Condition Variables*, which are the key mechanism for Monitors, as will be discussed.

5.5. Monitor Constructs

Monitors strictly speaking are a programming language feature. It means a compiler will identify keywords and ensure mutual exclusion. Originally they appeared on Concurrent Pascal, introduced by P.B. Hansen (curiously, the same computer scientist that coined the kernel concept, also regarded as a Monitor). *Java* also offers Monitors constructs.

But we can and do create Monitors in C, using kernel services. A Monitor is an ADT (data structure + functions);

5.5.1. Monitor Invariant

The key aspect of a monitor is its invariant:

> *The state of a Monitor will not change until the active task either sleeps or leaves the Monitor.*

This will become clearer when we see the implementation patterns.

5.5.2. Signalling Discipline

Within a Monitor the active task will *signal* one or all suspended tasks waiting for a condition it has recognised as being true. Now, this task has to give way for the waiter. There are three disciplines: signal-and-leave, signal-and-wait, signal-and-continue or: *Hansen*, *Hoare* and *Mesa Monitors*, respectively.

RK0 mechanisms are tailored for *Signal-and-continue (Mesa)* .

Signal-and-continue

If a signaller task holds a lock at the moment it signals a waiting queue, the signalled when resuming inside the monitor has to acquire the lock.

Thus, if the signaller is still active the signalled task will block. A task before going inactive needs to release the lock.

The drawback, is the waken task is delayed. By the time it finally acquires the lock, the condition

might have changed. For this reason, the test-loop has this pattern:

```
--- snippet ---
while (condition.isFalse())
{
    condition.wait();
    /* wake here, and
            re-check */
}
--- snippet ---
```

5.5.3. Producer-Consumer Solution using a Mesa Monitor

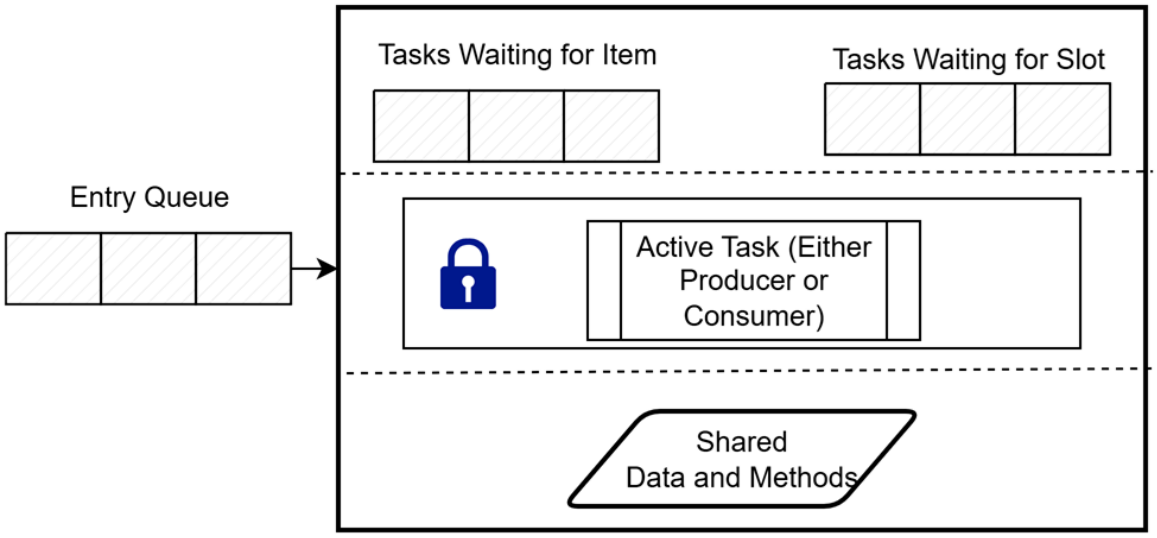
Here the, Producer Consumer Solution previously presented using semaphores is depicted on a monitor-like idiom.

```
/* MONITOR PSEUDO CODE */

/* this is an imaginary language */
Monitor BoundBuf
{
    /* pure data */
    UNSIGNED CONST  maxItems;
    UNSIGNED        currItems;

    /* other ADTs */
    LOCK        guard // monitor lock
    SLEEPQUEUE  noItem  // tasks waiting for an item
    SLEEPQUEUE  noSlot  // tasks waiting for a slot
    CIRCBUF     buf; // circular buffer

    /* Monitor methods */

    NORETURN InsertItem(ITEM_t* item)
    {

        guard.Lock(); /* lock monitor */

        /* condition for producers is having
        free slots */
        while (currItems == maxItems)  /* while (!notFull) */
        {
            /* is full */

            ENTER_CRITICAL_REGION

            guard.Unlock();

            noSlot.Wait();
```

```
            EXIT_CRITICAL_REGION

            /* when waking acquire mutex again */
            guard.Lock();
        }

        /* increase number of items */

        buf.Insert(item);

        currItems ++;

        /* signal any tasks waiting for an item */

        noItem.Signal();

        /* release monitor and leave */

        guard.Unlock();

    }

    ITEM_t* RemoveItem()
    {
        ......

    }

}
```

There is plenty happenning in the above pattern: the correct combination of locking, unlocking, sleeping and signalling under a predicate, is what enforce the correct precedence of tasks accessing a shared resource.

Note that locks are ordered to guarantee a single task owns the `currItems` variable. Importantly, when signalling a producer that the number of items decreased (or number of slots increased) even if the producer has a higher priority and is dispatched it will block trying to `guard.Lock()` when resuming within `while(!notFull)`.



The `UNLOCK-WAIT` sequence within the testing loop has preemption disabled because after releasing the lock, the task cannot be allowed to resume within the monitor again, for any reason that is not the monitor predicate being satisfied.

Lost signal?

What if the task increases `currItems` and is preempted before being able to signal a consumer? Because it is still holding the lock, it will resume within the monitor finding the same conditions when it has left. A logical consequence of the monitor invariant.

This is how *lost wake-ups* are prevented.

5.5.4. Condition Variable Model

The *Condition Variable Model* allows a task to wait within a monitor-construct and if active, operate using `signal()`, `wait()` and `broadcast()` respecting the monitor invariant.

In RK0, we handle Condition Variables with helper functions acting over a Mutex and a Sleep Queue:

- `kCondVarWait(&sleepq, &mutex, timeout)`
- `kCondVarSignal(&sleepq)`
- `kCondVarBroadcast(&sleepq)`

The `condWait` is the real helper. When using it, a Mesa testing-loop reduces to:

```
while(!condition)
{
  kCondVarWait(&condQueue, &monitorLock, timeout);
}
```

RK0 follows the same *Pthreads Condition Variables* semantics, which in turn are aligned with Mesa Monitors.

5.5.4.1. Usage Example: Synchronisation Barrier

A given number of tasks must reach a point in the program before *all can proceed*, so every task calls a `barrWait(&barrier)` to catch up with the set of tasks it must synchronise.

The last task entering the barrier will broadcast a signal to all tasks waiting for the wake condition.

At any moment within a Monitor a single task is RUNNING (what is an invariant of the kernel), all other tasks within the monitor are either SLEEPING (for some condition) or BLOCKED (on a mutex).

```
/* Synchronisation Barrier */

typedef struct
{
    RK_MUTEX lock;
    RK_SLEEP_QUEUE allSynch;
    UINT count;        /* number of tasks in the barrier */
    UINT round;        /* increased every time all tasks synch     */
    UINT nRequired; /* number of tasks required */



} Barrier_t;

VOID BarrierInit(Barrier_t *const barPtr, UINT nRequired)
{
    kMutexInit(&barPtr->lock, RK_INHERIT);
    kEventInit(&barPtr->allSynch);
    barPtr->count = 0;
    barPtr->round = 0;
    barPtr->Required = nRequired;

}

VOID BarrierWait(Barrier_t *const barPtr)
{
    UINT myRound = 0;
    kMutexLock(&barPtr->lock, RK_WAIT_FOREVER);

    /* save round number */
    myRound = barPtr->round;
    /* increase count on this round */
    barPtr->count++;

    if (barPtr->count == barPtr->nRequired)
    {
        /* reset counter, inc round, broadcast to sleeping tasks */
        barPtr->round++;
        barPtr->count = 0;
        kCondVarBroadcast(&barPtr->allSynch);
    }
```

```
    else
    {
        /* a proper wake signal might happen after inc round */
        while ((UINT)(barPtr->round - myRound) == 0U)
        {
            RK_ERR err = kCondVarWait(&barPtr->allSynch, &barPtr->lock,
RK_WAIT_FOREVER);
            K_ASSERT(err==RK_ERR_SUCCESS);
        }
    }

    kMutexUnlock(&barPtr->lock);

}


#define N_REQUIRED 3

Barrier_t syncBarrier;

VOID kApplicationInit(VOID)
{

    K_ASSERT(!kCreateTask(&task1Handle, Task1, RK_NO_ARGS, "Task1", stack1, STACKSIZE,
2, RK_PREEMPT));
    K_ASSERT(!kCreateTask(&task2Handle, Task2, RK_NO_ARGS, "Task2", stack2, STACKSIZE,
3, RK_PREEMPT));
    K_ASSERT(!kCreateTask(&task3Handle, Task3, RK_NO_ARGS, "Task3", stack3, STACKSIZE,
1, RK_PREEMPT));
    BarrierInit(&syncBarrier, N_REQUIRED);
}
VOID Task1(VOID* args)
{
    RK_UNUSEARGS
    while (1)
    {
        kPuts("Task 1 is waiting at the barrier...\r\n");
        BarrierWait(&syncBarrier);
        kPuts("Task 1 passed the barrier!\r\n");
        kSleep(8);

    }
}

VOID Task2(VOID* args)
{
    RK_UNUSEARGS
    while (1)
    {
        kPuts("Task 2 is waiting at the barrier...\r\n");
        BarrierWait(&syncBarrier);
```

```
        kPuts("Task 2 passed the barrier!\r\n");
        kSleep(5);
    }
}

VOID Task3(VOID* args)
{
    RK_UNUSEARGS
    while (1)
    {
        kPuts("Task 3 is waiting at the barrier...\r\n");
        BarrierWait(&syncBarrier);
        kPuts("Task 3 passed the barrier!\r\n");
        kSleep(3);
    }
}
```

```
Task 3 is waiting at the barrier...
Task 2 is waiting at the barrier...
Task 1 is waiting at the barrier...
Task 3 passed the barrier!
Task 1 passed the barrier!
Task 2 passed the barrier!
Task 3 is waiting at the barrier...
Task 2 is waiting at the barrier...
Task 1 is waiting at the barrier...
Task 3 passed the barrier!
Task 1 passed the barrier!
Task 2 passed the barrier!
Task 3 is waiting at the barrier...
Task 2 is waiting at the barrier...
Task 1 is waiting at the barrier...
Task 3 passed the barrier!
Task 1 passed the barrier!
Task 2 passed the barrier!
Task 3 is waiting at the barrier...
Task 2 is waiting at the barrier...
Task 1 is waiting at the barrier...
Task 3 passed the barrier!
Task 1 passed the barrier!
Task 2 passed the barrier!
```

5.5.4.2. Usage Example: Readers Writers Lock

Several readers and writers share a piece of memory. Readers can concurrently access the memory to read; a single writer is allowed (otherwise, data would be corrupted).

When a writer finishes, it checks for any readers waiting. If there is, the writer flushes the readers waiting queue. If not, it wakes a single writer, if any. When the last reader finishes, it signals a writer.

Every read or write operation begins with an acquire and finishes with a release.

*PS: This RWLock implementation has a **reader-preference** policy, as when a writer finishes, it flushes sleeping readers. When the last reader finishes, it will signal writer waiting queue.*

```
/* RW-Lock */

/* a single writer is allowed if there are no readers */
/* several readers are allowed if there is no writer*/
typedef struct
{
    RK_MUTEX     lock;
    RK_SLEEP_QUEUE   writersGo;
    RK_SLEEP_QUEUE   readersGo;
    INT          rwCount; /* number of active readers if > 0 */
                          /* active writer if -1             */

}RwLock_t;

VOID RwLockInit(RwLock_t *const rwLockPtr)
{

    kMutexInit(&rwLockPtr->lock, RK_INHERIT);
    kEventInit(&rwLockPtr->writersGo);
    kEventInit(&rwLockPtr->readersGo);
    rwLockPtr->rwCount = 0;
}

/* A writer can acquire if  rwCount = 0 */
/* An active writer is indicated by rwCount = -1; */
VOID RwLockAcquireWrite(RwLock_t *const rwLockPtr)
{
    kMutexLock(&rwLockPtr->lock, RK_WAIT_FOREVER);
    /* if different than 0, there are either writers or readers */
    /* sleep to be signalled */
    while (rwLockPtr->rwCount != 0)
    {
        kCondVarWait(&rwLockPtr->writersGo, &rwLockPtr->lock, RK_WAIT_FOREVER);
        /* mutex is locked when waking up*/
    }
    /* woke here, set an active writer */
    rwLockPtr->rwCount = -1;
    kMutexUnlock(&rwLockPtr->lock);
}

/* a writer releases, waking up all waiting readers, if any */
/* if there are no readers, a writer can get in */
VOID RwLockReleaseWrite(RwLock_t *const rwLockPtr)
{
    kMutexLock(&rwLockPtr->lock, RK_WAIT_FOREVER);

    rwLockPtr->rwCount = 0; /* indicate no writers*/

    /* if there are waiting readers, flush */
    ULONG nWaitingReaders=0;
```

```
    kEventQuery(&rwLockPtr->readersGo, &nWaitingReaders);
    if (nWaitingReaders > 0)
    {
        /* condVarBroadcast is just an alias for an event flush */
        kEventFlush(&rwLockPtr->readersGo);
    }
    else
    {
        /* wake up a single writer if any */
        kEventSignal(&rwLockPtr->writersGo);
    }
    kMutexUnlock(&rwLockPtr->lock);
}

/* a reader can acquire if there are no writers */
VOID RwLockAcquireRead(RwLock_t *const rwLockPtr)
{
    kMutexLock(&rwLockPtr->lock, RK_WAIT_FOREVER);
    /* if there is an active writer, sleep */
    while (rwLockPtr->rwCount < 0)
    {
        kCondVarWait(&rwLockPtr->readersGo, &rwLockPtr->lock, RK_WAIT_FOREVER);
        /* mutex is locked when waking up*/
    }
    /* increase rwCount, so its > 0, indicating readers */
    rwLockPtr->rwCount ++;
    kMutexUnlock(&rwLockPtr->lock);
}

/* a reader releases and wakes a single writer */
/* if it is the last reader */
VOID RwLockReleaseRead(RwLock_t *const rwLockPtr)
{
    kMutexLock(&rwLockPtr->lock, RK_WAIT_FOREVER);
    rwLockPtr->rwCount --;
    if (rwLockPtr->rwCount == 0)
    {
        kEventSignal(&rwLockPtr->writersGo);
    }
    kMutexUnlock(&rwLockPtr->lock);
}
```

In the image below, 4 tasks — a fast writer (Task 1), a slow writer (Task 4) and two readers (Task3 is faster than Task2) — reading from and writing to a shared UINT variable:

```
15340ms : Task 3 read 73
15390ms : Task 4 wrote 75
15500ms : Task 3 read 75
15550ms : Task 2 read 75
15620ms : Task 1 wrote 76
15750ms : Task 3 read 76
15910ms : Task 2 read 76
16000ms : Task 1 wrote 77
16100ms : Task 3 read 77
16200ms : Task 4 wrote 79
Task4 is flushing readers
16300ms : Task 3 read 79
16350ms : Task 2 read 79
16420ms : Task 1 wrote 80
16520ms : Task 3 read 80
16570ms : Task 4 wrote 82
Task4 is flushing readers
16670ms : Task 2 read 82
16750ms : Task 3 read 82
16800ms : Task 1 wrote 83
```

5.6. Message Queue

The key mechanism for Message Passing in RK0 is a *Message Queue*. Messages are passed *by copy* and have a fixed-size of 4-,8-,16- or 32-byte, i.e., 1, 2, 4 or 8 words.

Two other abstractions are constructed on top of Queues:

1. *Port*, which is a server endpoint that enqueues *Remote Invocations*, or *Procedure Calls* from clients.
2. *Mailbox*, a 1-word message queue. It can be used as is, and is used by the kernel as reply route from servers to clients.

Message Queue Control Block
Buffer Address
Message Size
Number of Mesages
Write Position
Read Position
Server Status
Owner Task
Notify callback
Waiting queue

5.6.1. Size of a Message

Each declared queue has a *fixed message-size* at initialisation, and can assume, 1, 2, 4 or 8 WORDs (4, 8, 16, 32 BYTEs). This constraint is intentional. Word-aligned copies are faster, predictable and safer for type casting.

(A word-aligned *single copy* will take ~5 cycles in Cortex-M3/4/7, and ~6 cycles on Cortex-M0/M0+.)

Ports have messages whose types are labeled as:

- `RK_PORT_MESG_<2,4,8>WORD`, which the first two are reserved meta-data. For variable payload one uses the `RK_PORT_MESG_COOKIE`, passing an opaque spointer as payload.

5.6.2. Blocking and non-blocking behaviour

A producer task can optionally block on a full queue, switching its state to `SENDING`. A consumer (optionally) blocks on an empty queue switching its state to `RECEIVING`.

Note that when using non-blocking calls (`RK_NO_WAIT`) they immediately return if unsuccessful.

One can use a `peek()` to read from the head of a queue without extracting the message. A `jam()` method is used to put a message on the queue head, but it does not overwrite. `jam()` is meaningless for Mailboxes.

A `postovw()` overwrites the oldest message and is allowed solely for 1-message queues. If N-message queues were allowed to be overwritten, all unread messages would leak as read and write pointers are adjusted on the operation. In case one needs overwriting full queues continuously a classic ring buffer will do.

`peek()` and `postovw()` are normally used on Mailboxes for last-message semantics — the Mailbox never goes empty; a new message is placed by overwriting the current one.

5.6.3. Ownership and Priority Inversion

A remark of the Message Passing mechanism in *RK0* is that it handles priority inversion.

Message Queues (an by extension, Mailboxes) *can* have owners, and thus, only the owners can receive at these channels. One might think it makes it a *Port*, but not yet. The fact is, if a receiver is blocked on full queue, the sender can boost its priority.

On the other hand, *Ports* are for _ synchronous procedure calls_. The server needs to run at the client's priority, which means either boosting or demoting priority.

5.6.4. Notify callback

A callback can be registered for when a queue sends a message successfuly, as a means of event notification.

5.6.5. Usage Examples

Below there is a rich set of usage examples. They demonstrate how to use message-passing in RK0, its API and helper macros to declare storage and messages at the appropriate size.

5.6.5.1. Averaging Sensor Values

A task receives measured sensor values from an ISR on a periodic rate. (A Soft Timer emulates the ISR).

Then it enqueues this data to a consumer - that will process the average value for each of 4 sensors.

The inter-task communication is designed as follows:

1. The producer pends on a Mailbox that an ISR posts to. An application timer emulates this ISR.
2. The data extracted from the Mailbox is placed in a queue with the processing task as the consumer.
3. As the producer's priority must be higher than that of the consumer, eventually, the queue will get full.
4. The producer drops the last message when the queue is full and signals the consumer.
5. Now the consumer has a batch of data to work until the next sensor update. It will block (pend on a signal) whenever the queue is empty.

```
/* helpers for pend and signal */

#define kPend(timeout)                                  \
    do                                                  \
    {                                                   \
        kTaskFlagsGet(0x1, RK_FLAGS_ANY, NULL, timeout); \
    } while (0)

#define kSignal(taskhandle)           \
    do                                \
    {                                 \
        kTaskFlagsSet(taskhandle, 0x01); \
    } while (0)


/* sensor type */
typedef enum
{
    TEMPERATURE=1, HUMIDITY, CO2, FLOW
}SensorType_t;

struct sensorMsg
{
    SensorType_t sensorType;
    ULONG sensorValue;
```

```
};
typedef struct sensorMsg Mesg_t;

#define N_SENSOR    4
#define AVG_WINDOW_SIZE   10 /* 10 samples */

/* the queue */
RK_MESG_QUEUE sensorStream;
/* convenience macro to declare the queue storage */
#define N_MESSAGE 8
RK_DECLARE_MESGQ_BUF(mesgBuf, Mesg_t, N_MESSAGE)

/* timer to mimic isr */
RK_TIMER timerT1;

/* the mailbox the sensor task pends */
RK_MAILBOX sensorBox;
static Mesg_t sample = {0};
static UINT sampleErr;
VOID callBackISR(VOID* args);

VOID kApplicationInit( VOID)
{
    K_ASSERT(!kMesgQueueInit(&sensorStream,
                               mesgBuf,
                            RK_MESGQ_MESG_SIZE(Mesg_t), /* set mesg size */
                               N_MESSAGE));
    /* timer @ every 10 ms */
    K_ASSERT(!kTimerInit(&timerT1, 0, 10, callBackISR, NULL, RK_TIMER_RELOAD));
    K_ASSERT(!kMailboxInit(&sensorBox));
}

VOID callBackISR(VOID *args)
{
    RK_UNUSEARGS
    sample.sensorType = (rand() % 4) + 1;
    switch (sample.sensorType)
    {
        case TEMPERATURE:
            sample.sensorValue = ( ULONG) rand() % 50;
            break;
        case HUMIDITY:
            sample.sensorValue = ( ULONG) rand() % 100;
            break;
        case CO2:
            sample.sensorValue = ( ULONG) rand() % 1000;
            break;
        case FLOW:
            sample.sensorValue = ( ULONG) rand() % 10;
            break;
        default:
```

```
            break;
    }
    /* Mailbox carries one word: post a pointer to sample */
    Mesg_t *samplePtr = &sample;
    RK_ERR err = kMailboxPost(&sensorBox, &samplePtr, RK_NO_WAIT);
    if (err != RK_ERR_SUCCESS)
        sampleErr ++;

}

/* Producer - higher priority, blocks on mailbox */
VOID Task1(VOID *args)
{
    RK_UNUSEARGS
    Mesg_t *recvSample = NULL;
    while (1)
    {
        /* receive a pointer */
        RK_ERR errmbox = kMailboxPend(&sensorBox, &recvSample,RK_WAIT_FOREVER);
        K_ASSERT( errmbox==RK_ERR_SUCCESS);

        /* enqueue by copy into the stream */
        RK_ERR err = kMesgQueueSend(&sensorStream, recvSample, RK_NO_WAIT);

        K_ASSERT(err >= 0); /* either succesful or unsuccesful */
        if (err == RK_ERR_SUCCESS)
        {
            CHAR const *sensorTypeStr = NULL;
            if (recvSample->sensorType == 1)
                sensorTypeStr = "TEMP";
            if (recvSample->sensorType == 2)
                sensorTypeStr = "HUM";
            if (recvSample->sensorType == 3)
                sensorTypeStr = "CO2";
            if (recvSample->sensorType == 4)
                    sensorTypeStr = "FLOW";
            printf( "ENQ: [@%lums, %s, %lu] \r\n", kTickGet(), sensorTypeStr,
                        recvSample->sensorValue);
        }
        /* full, drop this sample and signal task2 */
        else if (err == RK_ERR_MESGQ_FULL)
        {
            kSignal(task2Handle);
        }
    }
}

/* for each sensor:
     . a ring buffer of AVG_WINDOW_SIZE values
 . sum of values
 . an index table (=enum - 1 eg., HUMIDITY IDX=2-1=1)
```

```
 */
static ULONG ringBuf[N_SENSOR][AVG_WINDOW_SIZE];
static ULONG ringSum[N_SENSOR] = {0};
static UINT ringIndex[N_SENSOR] = {0};

void Task2( void *args)
{

    RK_UNUSEARGS
    Mesg_t readSample;
    while (1)
    {
        RK_ERR err = kMesgQueueRecv(&sensorStream, &readSample, RK_NO_WAIT);
        if (err == RK_ERR_SUCCESS)
        {
            UINT sensorIdx = readSample.sensorType - 1;

/* remove oldest sample */
            ULONG oldest = ringBuf[sensorIdx][ringIndex[sensorIdx]];
            ringSum[sensorIdx] -= oldest;

/* push new sample */
            ringBuf[sensorIdx][ringIndex[sensorIdx]] = readSample.sensorValue;
            ringSum[sensorIdx] += readSample.sensorValue;

/* index incr-wrap */
            ringIndex[sensorIdx] ++;
            ringIndex[sensorIdx] %= AVG_WINDOW_SIZE;

/* simple average */
            ULONG avg = ringSum[sensorIdx] / AVG_WINDOW_SIZE;


            CHAR const *sensorTypeStr = NULL;
            if (readSample.sensorType == 1)
                sensorTypeStr = "TEMP";
            if (readSample.sensorType == 2)
                sensorTypeStr = "HUM";
            if (readSample.sensorType == 3)
                sensorTypeStr = "CO2";
            if (readSample.sensorType == 4)
                sensorTypeStr = "FLOW";

            printf( "DEQ: [@%lums, %s, %lu] | AVG: %lu \r\n", kTickGet(),
                    sensorTypeStr, readSample.sensorValue, avg);

        }
        else
        {
            kPend(RK_WAIT_FOREVER);
        }
```

```
    }
}
```

```
OUTPUT:

ENQ: [@550ms, CO2, 571]
ENQ: [@560ms, FLOW, 4]
ENQ: [@570ms, FLOW, 4]
ENQ: [@580ms, HUM, 25]
ENQ: [@590ms, CO2, 931]
ENQ: [@600ms, CO2, 487]
ENQ: [@610ms, FLOW, 7]
ENQ: [@620ms, HUM, 79]

>>> Queue is full. Now offload and process. Note the order remains <<<

DEQ: [@630ms, CO2, 571] | AVG: 460
DEQ: [@631ms, FLOW, 4] | AVG: 5
DEQ: [@632ms, FLOW, 4] | AVG: 5
DEQ: [@633ms, HUM, 25] | AVG: 52
DEQ: [@634ms, CO2, 931] | AVG: 553
DEQ: [@635ms, CO2, 487] | AVG: 549
DEQ: [@636ms, FLOW, 7] | AVG: 5
DEQ: [@637ms, HUM, 79] | AVG: 55

>>> Consumer is preempted <<<
ENQ: [@640ms, CO2, 913]
ENQ: [@650ms, CO2, 134]
ENQ: [@660ms, HUM, 47]
ENQ: [@670ms, HUM, 30]
ENQ: [@680ms, TEMP, 7]
ENQ: [@690ms, CO2, 726]
ENQ: [@700ms, FLOW, 7]
ENQ: [@710ms, TEMP, 43]

DEQ: [@720ms, CO2, 913] | AVG: 578
DEQ: [@721ms, CO2, 134] | AVG: 543
DEQ: [@722ms, HUM, 47] | AVG: 51
DEQ: [@723ms, HUM, 30] | AVG: 44
DEQ: [@724ms, TEMP, 7] | AVG: 20
DEQ: [@725ms, CO2, 726] | AVG: 592
DEQ: [@726ms, FLOW, 7] | AVG: 5
DEQ: [@727ms, TEMP, 43] | AVG: 23
```

5.6.5.2. Queue Select using Notify Callback

It is not uncommon to have a gatekeeper or supervisor task listening to several queues at once.

In this snippet, a `sendNotify` is installed on for each queue that signal the supervisor task. This task runs every 100ms coalescing about two `post` of each mail queue.

Note the information sent through the queue is the address of a memory block on a partition pool. The sender allocates and consumer frees. The sender frees if finding an empty poool.

```
/* Notify Callback on Queues */

/* each queue has registered this send callback */
VOID sendNotify(RK_MESG_QUEUE *qPtr)
{
    UINT i = 0;
    for (i = 0; i < 3; ++i)
    {
        if (queues[i] == qPtr)
        {
            ULONG qFlag = 1UL << i;
            kTaskFlagsSet(superHandle, qFlag);
            break;
        }
    }
}

/* tasks sending follow this pattern */

VOID Task2(VOID *args)
{
    RK_UNUSEARGS

    UINT num = 0x20;

    while (1)
    {
        MESG_t *ptr;
        /* allocate pointer */
        ptr = (MESG_t *)kMemPartitionAlloc(&queueMem);
        if (ptr)
        {
            ptr->num = num++;
            ptr->senderID = RK_RUNNING_PID;
            RK_ERR err = kMesgQueueSend(&queue2, &ptr, RK_NO_WAIT);
            if (err != RK_ERR_SUCCESS)
            {
                kMemPartitionFree(&qMem, ptr);
                kprintf("Q2 FULL\n\r");
            }
         }
        kSleepPeriod(50);
    }
}
```

```
/* supervisor listening on 3 queues */

/*
although it has a higher period, it has also the highest priority, what is acceptable
for supervisors
you see the tasks posting to queues are keeping its two succesfull posts the
supervisor drains
every 100ms
*/

VOID SupervisorTask(VOID *args)
{    RK_UNUSEARGS

    static ULONG gotFlags = 0UL;

    while (1)
    {
        gotFlags = 0UL;
        kTaskFlagsGet(0x7, RK_FLAGS_ANY, &gotFlags, RK_NO_WAIT);

        UINT k = 0;

        /* we drain each queue that is set */
        for (k = 0; k < 3; ++k)
        {
            if (gotFlags & (1UL << k))
            {
                VOID *recvPtr = NULL;

                while (
                    kMesgQueueRecv(queues[k], &recvPtr,
                                RK_NO_WAIT) == RK_ERR_SUCCESS)
                {
                    MESG_t *m = (MESG_t*)recvPtr;
                    /*1-copy message passing */
                    UINT id = m->senderID;
                    UINT num = m->num;

                    kMemPartitionFree(&queueMem, recvPtr);

                    UINT sel = k + 1; /* bit position */
                    logPost(" sel: %d, senderID: %d, payload: 0x%02X", sel, id, num);

                }
            }
        }
        kSleepPeriod(100);
    }
}
```

```
OUTPUT:
36500 ms ::  sel: 1, senderID: 5, payload: 0x2E8
36500 ms ::  sel: 1, senderID: 5, payload: 0x2E9
36500 ms ::  sel: 2, senderID: 3, payload: 0x2F8
36500 ms ::  sel: 2, senderID: 3, payload: 0x2F9
36500 ms ::  sel: 3, senderID: 4, payload: 0x308
36500 ms ::  sel: 3, senderID: 4, payload: 0x309
36600 ms ::  sel: 1, senderID: 5, payload: 0x2EA
36600 ms ::  sel: 1, senderID: 5, payload: 0x2EB
36600 ms ::  sel: 2, senderID: 3, payload: 0x2FA
36600 ms ::  sel: 2, senderID: 3, payload: 0x2FB
36600 ms ::  sel: 3, senderID: 4, payload: 0x30A
36600 ms ::  sel: 3, senderID: 4, payload: 0x30B
36700 ms ::  sel: 1, senderID: 5, payload: 0x2EC
36700 ms ::  sel: 1, senderID: 5, payload: 0x2ED
36700 ms ::  sel: 2, senderID: 3, payload: 0x2FC
36700 ms ::  sel: 2, senderID: 3, payload: 0x2FD
36700 ms ::  sel: 3, senderID: 4, payload: 0x30C
36700 ms ::  sel: 3, senderID: 4, payload: 0x30D
36800 ms ::  sel: 1, senderID: 5, payload: 0x2EE
36800 ms ::  sel: 1, senderID: 5, payload: 0x2EF
36800 ms ::  sel: 2, senderID: 3, payload: 0x2FE
36800 ms ::  sel: 2, senderID: 3, payload: 0x2FF
36800 ms ::  sel: 3, senderID: 4, payload: 0x30E
36800 ms ::  sel: 3, senderID: 4, payload: 0x30F
36900 ms ::  sel: 1, senderID: 5, payload: 0x2F0
36900 ms ::  sel: 1, senderID: 5, payload: 0x2F1
36900 ms ::  sel: 2, senderID: 3, payload: 0x300
```

5.6.5.3. Synchronous Client-Server Procedure Call

The example below computes a CRC on the server for the client's payload and returns it as the reply code. Logically, this is unbuffered: the client blocks until it gets the reply.

Note the server has its priority demoted while serving the the client. Upon finishing its priority is restored.

```
#include <application.h>
#include <logger.h>
define STACKSIZE 256
#define PORT_MSG_WORDS 4U   /* 2 words meta + 2 words payload */
#define PORT_CAPACITY 16

/* tasks */
RK_DECLARE_TASK(serverHandle, ServerTask,    stack1, STACKSIZE)
RK_DECLARE_TASK(clientHandle, ClientTask,    stack2, STACKSIZE)

/* port */
static RK_PORT serverPort;
```

```
RK_DECLARE_PORT_BUF(portBuf, PORT_MSG_WORDS, PORT_CAPACITY)

/* 4-word message format; first two are reserved
for senderID and reply address */
typedef RK_PORT_MESG_4WORD RpcMsg;

static inline UINT crc32(const VOID *data, ULONG size);
static inline BYTE xorshift8(void);

VOID kApplicationInit(void)
{
    K_ASSERT(!kCreateTask(&serverHandle, ServerTask, RK_NO_ARGS,
                          "Server", stack1, STACKSIZE, 1, RK_PREEMPT));
    K_ASSERT(!kCreateTask(&clientHandle, ClientTask, RK_NO_ARGS,
                          "Client", stack2, STACKSIZE, 2, RK_PREEMPT));

    /* init port */
    K_ASSERT(!kPortInit(&serverPort, portBuf, PORT_MSG_WORDS, PORT_CAPACITY,
serverHandle));

    logInit();
}

VOID ServerTask(VOID *args)
{
    RK_UNUSEARGS
    RpcMsg msg;
    while(1)
    {
        /* receive next request; server may adopt client priority here */
        K_ASSERT(!kPortRecv(&serverPort, &msg, RK_WAIT_FOREVER));

        BYTE  *vector = (BYTE*) msg.payload[0];
        ULONG   size  =          msg.payload[1];
        UINT    crc   = crc32(vector, size);

        logPost("[SERVER] Will Reply CRC=0x%04X | Eff Prio=%d | Real Prio=%d",
                crc, runPtr->priority, runPtr->prioReal);

        /* must end with kPortReplyDone */
        K_ASSERT(!kPortReplyDone(&serverPort, (ULONG const*)&msg, crc));

        logPost("[SERVER] Finished. | Eff Prio: %d | Real Prio: %d", runPtr->priority,
runPtr->prioReal);
    }
}

VOID ClientTask(VOID *args)
{
    RK_UNUSEARGS
    static BYTE vec[8];
```

```
    for (UINT i = 0; i < 8; ++i)
        vec[i] = xorshift8();

    RK_MAILBOX replyBox;
    kMailboxInit(&replyBox);
    RpcMsg msg = {0};
    msg.payload[0] = (ULONG) vec;  /* pointer as one word */
    msg.payload[1] = 8;            /* number of bytes */

    UINT reply = 0;
    while(1)
    {
        /* Send-Receive: a call */
        UINT want = crc32(vec, 8);
        K_ASSERT(!kPortSendRecv(&serverPort, (ULONG*)&msg, &replyBox, &reply,
                                RK_WAIT_FOREVER));
        logPost("[CLIENT] Need=0x%04X | Recvd=0x%04X", want, reply);
        /* if reply is correct, generate a new payload */
        if (want == reply)
            for (UINT i = 0; i < 8; ++i) vec[i] = xorshift8();
        kSleepPeriod(1000);
    }
}
```

```
   28000 ms :: [CLIENT] Need=0xC2A6C337 | Recvd=0xC2A6C337
   29000 ms :: [SERVER] Will Reply CRC=0x93F4110A | Eff Prio=2 | Real Prio=1
   29000 ms :: [SERVER] Finished. | Eff Prio: 1 | Real Prio: 1
   29000 ms :: [CLIENT] Need=0x93F4110A | Recvd=0x93F4110A
   30000 ms :: [SERVER] Will Reply CRC=0x7A8FA006 | Eff Prio=2 | Real Prio=1
   30000 ms :: [SERVER] Finished. | Eff Prio: 1 | Real Prio: 1
   30000 ms :: [CLIENT] Need=0x7A8FA006 | Recvd=0x7A8FA006
   31000 ms :: [SERVER] Will Reply CRC=0x9051C8B1 | Eff Prio=2 | Real Prio=1
   31000 ms :: [SERVER] Finished. | Eff Prio: 1 | Real Prio: 1
   31000 ms :: [CLIENT] Need=0x9051C8B1 | Recvd=0x9051C8B1
   32000 ms :: [SERVER] Will Reply CRC=0x11F29117 | Eff Prio=2 | Real Prio=1
   32000 ms :: [SERVER] Finished. | Eff Prio: 1 | Real Prio: 1
```

5.7. Most-Recent Message Protocol (MRM)

MRM Control Block
MRM Buffer Allocator
Data Buffer Allocator
Current MRM Buffer Address
Data Size (Message Size)

MRM Buffer
Data Buffer Address
Readers Count

Data Buffer
Application-dependent

There is little practical difference between a message that does not arrive and one with no valid (stale) data. But when wrong (or stale) data is processed - e.g., to define a set point on a loop - a system can fail badly.

Design Choice: provide a broadcast asynchronous message-passing scheme that guarantees data freshness and integrity for all readers.

Benefits: The system has a mechanism to meet strict deadlines that cannot be predicted on design time.

Control loops reacting to unpredictable time events—like a robot scanning an environment or a drive-by-wire system—require a different message-passing approach. Readers cannot "look at the past" and cannot block. The most recent data must be delivered lock-free and have guaranteed integrity.

5.7.1. Functional Description

An *MRM* works as a *1-to-many asynchronous Mailbox* - with a lock-free specialisation that enables several readers to get the most recent deposited message with no integrity issues. Whenever a reader reads an MRM buffer, it will find the most recent data transmitted. It can also be seen as an extension of the Double Buffer pattern for a 1:N communication.

The core idea of the MRM protocol is that readers can only access the buffer that is classified as the '*most recent buffer*'. After a writer *publish()* a message, that will be the only message readers can *get()* — any former message being processed by a reader was grabbed *before* a new *publish()* - and, from now on, can only be *unget()*, eventually returning to the pool.

To clarify further, the communication steps are listed:

1. A producer first reserves an MRM Buffer - the reserved MRM Buffer is not available for reading until it is published.
2. A message buffer is allocated and filled, and its address is within an MRM Buffer. The producer *publishes* the message. From now on, it is *the most recent message*. Any former published buffer is no longer visible to new readers
3. A reader starts by *getting* an MRM Buffer. A `get()` operation delivers a copy of the message to the reader's scope. Importantly, this operation increases the number of readers associated to that MRM Buffer.

Before ending its cycle, the task releases (`unget()`) the buffer; on releasing, the kernel checks if the caller task is the last reader and if the buffer being released is not the current MRM Buffer.

If the above conditions are met, the `unget()` operation will return the MRM buffer to the pool. If there are more readers, OR if it is the current buffer, it remains available.

When the `reserve` operation detects that the most recent buffer still has readers, a new buffer is allocated to be written and published. If it has no readers, it is reused.

This way, the worst case is a sequence of `publish()` with no `unget()` at all — this would lead to the writer finding no buffer to reserve. This is prevented by making: `N Buffers = N tasks + 1`.

5.7.2. MRM Control Block Configuration

What might lead to some confusion when initialising an MRM Control Block is the need for two different pools:

- One pool will be the storage for the MRM Buffers, which is the data structure for the mechanism.
- Another pool is for the actual payload. The messages.

Both pools must have the same number of elements: the number of tasks communicating + 1.

- The size of the data buffers is application-dependent - and is passed as a number of *words*. The minimal message size is 32-bit.
- If using data structures, keep it aligned to 4 to take advantage of the performance of aligned memory.

5.7.3. Usage Example

Consider a modern car - speed variations are of interest in many modules. With a somehow "naive" approach, let us consider three modules and how they should react when speed varies:

1. **Cruiser Control:** For the Cruiser Control, a speed increase might signify the driver wants manual control back, and it will likely turn off.
2. **Windshield Wipers:** If they are on, a speed change can affect the electric motor's adjustments to the air resistance.
3. **Radio:** Speed changes reflect the aerodynamic noise - the radio volume might need adjustment.

As the variations are unpredictable, we need a mechanism to deliver the last speed in order of importance for all these modules. From highest to lowest priority, Cruise, Wipers, and Radio are the three modules that range from safety to comfort.

To emulate this scenario, we can write an application with a higher priority task that sleeps and wakes up at pseudo-random times to produce random values that represent the (unpredictable) speed changes.

The snippet below has 4 periodic tasks. Tasks sleep for absolute periods. The producer publishes new data at a random interval, so it can either interrupt before one has the chance to finish or be inactive while it runs more than once.

```
typedef struct
{
    UINT    speed;
    RK_TICK timeStamp;
} Mesg_t;

#define N_MRM (5)  /* Number of MRMs N Tasks + 1 */
#define MRM_MESG_SIZE (sizeof(Mesg_t)/4) /* In WORDS */
RK_MRM MRMCtl;/* MRM control block */
RK_MRM_BUF buf[N_MRM];/* MRM pool */
Mesg_t data[N_MRM];/* message data pool */


VOID kApplicationInit( VOID)
{
    kCreateTask(&speedSensorHandle, SpeedSensorTask, RK_NO_ARGS, "SpeedTsk",
stack1, STACKSIZE, 1, RK_PREEMPT);
    kCreateTask(&cruiserHandle, CruiserTask, RK_NO_ARGS, "CruiserTsk", stack2,
STACKSIZE, 2, RK_PREEMPT);
    kCreateTask(&wiperHandle, WiperTask, RK_NO_ARGS, "WiperTsk", stack3,
STACKSIZE, 3, RK_PREEMPT);
    kCreateTask(&radioHandle, RadioTask, RK_NO_ARGS, "RadioTsk", stack4,
STACKSIZE, 4, RK_PREEMPT);
    kMRMInit( &MRMCtl, buf, data, N_MRM, MRM_MESG_SIZE);
 }

VOID SpeedSensorTask( VOID *args)
{
    RK_UNUSEARGS

    Mesg_t sendMesg = {0};
    while (1)
    {
        RK_TICK currTick = kTickGet();
        UINT speedValue = (UINT) (rand() % 170) + 1;
        sendMesg.speed = speedValue;
        sendMesg.timeStamp = currTick;
        /* grab a buffer */
        RK_MRM_BUF *bufPtr =  kMRMReserve( &MRMCtl);
        if (bufPtr != NULL)
        {
            kMRMPublish( &MRMCtl, bufPtr,  &sendMesg);
        }
        else
        {/* cannot fail */
            K_ASSERT( 0);
        }
/* publish  */
         printf( "! @ %dT: SPEED UPDATE: %u \r\n", currTick, speedValue);
```

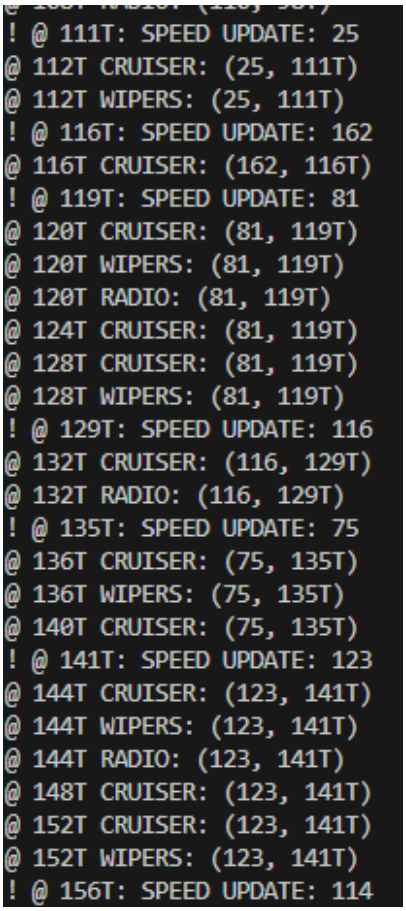
```
        RK_TICK sleepTicks = (( RK_TICK) rand() % 15) + 1;
        kSleepPeriod( sleepTicks);
    }
}

VOID CruiserTask( VOID *args)
{
    RK_UNUSEARGS

    Mesg_t recvMesg = {0};
    while (1)
    {
        RK_MRM_BUF *readBufPtr = kMRMGet( &MRMCtl,  &recvMesg);
        printf( "@ %dT CRUISER: (%u, %uT) \r\n", kTickGet(), recvMesg.speed,
recvMesg.timeStamp);
        kMRMUnget( &MRMCtl, readBufPtr);
        kSleepPeriod( 4);

    }
}


VOID WiperTask( VOID *args)
{
    RK_UNUSEARGS
    Mesg_t recvMesg = {0};

    while (1)
    {

        RK_MRM_BUF *readBufPtr = kMRMGet( &MRMCtl,  &recvMesg);
        printf( "@ %dT WIPERS: (%u, %uT) \r\n", kTickGet(), recvMesg.speed,
recvMesg.timeStamp);
        kMRMUnget( &MRMCtl, readBufPtr);
        kSleepPeriod( 8);


    }
}
VOID RadioTask( VOID *args)
{
    RK_UNUSEARGS
    Mesg_t recvMesg = {0};

    while (1)

    {
        RK_MRM_BUF *readBufPtr = kMRMGet( &MRMCtl, &recvMesg);
        printf( "@ %dT RADIO: (%u, %uT) \r\n", kTickGet(), recvMesg.speed,
recvMesg.timeStamp);
        kMRMUnget( &MRMCtl, readBufPtr);
```

```
        kSleepPeriod(12);

    }

}
```

Thus, different situations can happen:

- All tasks read the updated pair (speed, time)
- Not all tasks receive the updated pair because another update happens in between.
- No tasks receive an update - because another happens too soon.
- All tasks receive an update and will keep rereading the same values.

All these cases are on the image:



The highlight is that controllers can keep their pace, while receiving fresh data - you can see it on the timestamp on the image. Again, they might receive the same data more than once or miss samples; what is important is that *they are not lagging and consuming stale data.*

Chapter 6. *Error Handling*

6.1. Fail fast

While tracing and error handling are yet to be largely improved (and that is when the 1.0.0 version will be released), currently *RK0* employs a policy of *failing fast* in **debug mode**.

When Error Checking is enabled, every kernel call will be 'defensive', checking for correctness of parameters and invariants, null dereferences, etc.

In these cases is more useful to allow the first error to halt the execution by calling an Error Handler function to observe the program state.

A trace structure records the address of the running TCB, its current stack pointer, the link register (that is, the PC at kErrHandler was called), and a time stamp.

This record is on a `.noinit` RAM section, so it is visible if CPU resets. A fault code is stored in a global `faultID` and on the trace structure. Developers can hook in custom behaviour.

If the kernel is configured to not halt on a fault, but Error Checking is enabled, functions will return negative values in case of an error.

On the other hand, when Error Checking is disabled or `NDEBUG` is defined nothing is checked, reducing code size and improving performance.

(*Some deeper internal calls have assertion. For those, only* `NDEBUG` *defined ensures they are disabled.*)

6.2. Stack Overflow

Stack overflow is detected (not prevented) using a "stack painting" with a sentinel word. Stack Overflow detection is enabled by defining the assembler preprocessor `__KDEF_STACKOVFLW` when compiling.

One can take advantage of the static task model - *it is possible to predict offline* the deepest call within any task. The `-fstack-usage compiler` flag will create `.su` files indicating the depth of every function within a module. This is an example for the results of a compilation unit:

```
core/src/ksynch.c:61:8:kSignalGet   112 static
core/src/ksynch.c:163:8:kTaskFlagsSet   72  static
core/src/ksynch.c:214:8:kSignalClear    40  static
core/src/ksynch.c:237:8:kSignalQuery    56  static
core/src/ksynch.c:260:8:kEventInit  40  static
core/src/ksynch.c:279:8:kEventSleep 112 static
core/src/ksynch.c:343:8:kEventWake  88  static
core/src/ksynch.c:383:8:kEventSignal    64  static
core/src/ksynch.c:412:7:kEventQuery 16  static
core/src/ksynch.c:429:8:kSemaphoreInit  88  static
core/src/ksynch.c:470:8:kSemaphorePend  96  static
```

```
core/src/ksynch.c:555:8:kSemaphorePost  88  static
core/src/ksynch.c:603:8:kSemaphoreWake  72  static
core/src/ksynch.c:640:5:kSemaphoreQuery 40  static
core/src/ksynch.c:662:8:kMutexInit  16  static
core/src/ksynch.c:681:8:kMutexLock  120 static
core/src/ksynch.c:761:8:kMutexUnlock    96  static
core/src/ksynch.c:834:6:kMutexQuery 56  static
```

These are the worst cases. Now, you identify the depth of the longest *chain of calls* for a task using these services and add a generous safety margin — 30%. The cap depends on your budget.

Importantly, you also have to size the System Stack. This initial size is defined on `linker.ld` and on the symbol `Min_Stack_Size`. In this case, you need to account for the depth of `main()`, `kApplicationInit()`, and all interrupt handlers — again, inspect the the longest call chain depth. Assume interrupts will always add to the worst static depth, and make sure to account for *nested interrupts*.

6.3. Deadlocks

There are deadlock recovering mechanisms in the literature, a pity they are unfeasible here. The kernel provides bounded waiting, enforces every waiting queue to priority discipline, and applies priority inheritance to mutexes and message passing. Besides, it provides lock-free primitives and compensates for time drifting if a period is enforced on tasks. Well, none of these techniques can prevent deadlocks (right, with bounded blocking and lock free primitives one can get *livelocks*).

- Ordered Locking:

For those programming the application, despite following the RMS rule of higher priority for higher request rate tasks, the golden rule for locking is acquiring resources in an unidirectional order *throughout the entire application*:

```
acquire(A);
acquire(B);
acquire(C);
   .
   .
   .
release(C);
release(B);
release(A);
```

This breaks circular waiting.

For instance:

```
TaskA:
   wait(R1);
   wait(R2);
```

```
    /* critical section */
   signal(R2);
   signal(R1);

TaskB:
   wait(R1);
   wait(R2);
    /* critical section */
   signal(R2);
   signal(R1);
```

But, if:

```
TaskA:
    wait(R1);
    wait(R2);
    .
    .

TaskB:
    wait(R2);
    wait(R1);
    .
    .
```

There are some possible outcomes:

1. Deadlock:
    - TaskA runs: acquires R1
    - TaskB runs: acquires R2
    - TaskA runs: tries to acquire R2 — blocked
    - TaskB runs: tries to acquire R1 — blocked
2. No deadlock:
    - TaskA runs: acquires R1
    - TaskA runs: acquires R2 (nobody is holding R2)
    - TaskA releases both; TaskB runs and acquires both (in either order)

Overall, there is no deadlock if tasks do not overlap in critical sections. That is why systems run for years without deadlocks and eventually: *ploft*.

- Use a 'master-lock' with 'try' semantics:

Another technique that can be employed is if one needs to acquire multiple locks—acquire them all or none using a try-lock (`RK_NO_WAIT`). If any of the tries fail, the task gives up on acquiring the resources and backs off, releasing all successful locks to retry later (most simply, using a sleep

queue). That is easier said than done, though, and, as mentioned, if not well done, instead of deadlocks, one gets livelocks.

(*Livelocks* are when a couple of tasks keep running, but the program does not advance.)

Chapter 7. *RK0 Services API*

Convention

- A kernel call starts with a lowercase `k`. Typically it is followed by a kernel object identifier and an action.

```
kSemaphorePend(&sema, 800); /* pend on a semaphore; 800 ticks time-out */
```

- When `k` is followed by an action, it is acting on the caller task.

```
kSleep(150); /* sleep-delay the caller task for 150 ticks */
```

- Some calls can act either on the caller or on another task:

```
/* stores the signal flags of the task identified by task1Handle on queryValue */
kTaskFlagsQuery(task1Handle, &queryValue);

/* retrieves its own signal flags */
kTaskFlagsQuery(NULL, &queryValue);
```

Return Values

With a few exceptions, kernel calls return a `RK_ERR` error code. `0` is a successful operation `(RK_SUCCESS)` and any negative value is an error that indicates failure. A positive value is an unsuccessful operation, but will not lead the system to failure (e.g., any unsuccessful `try` operation).

- Find the most up to date RK0 API at the repo: RK0 API